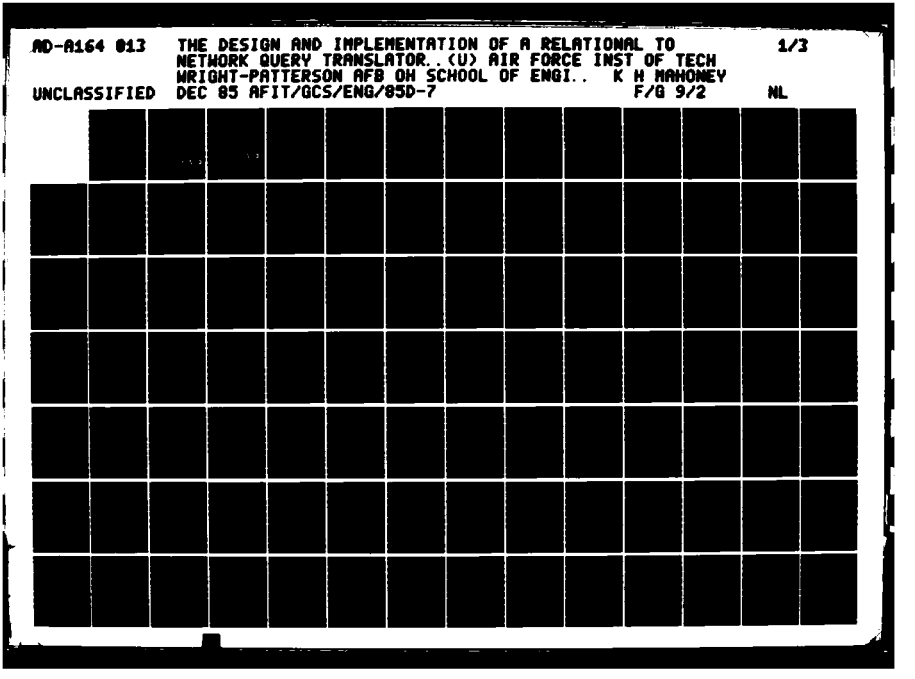
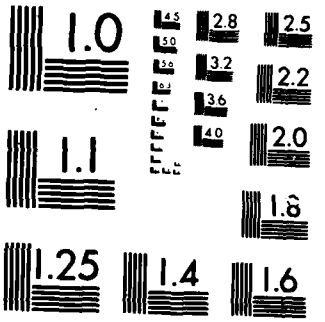
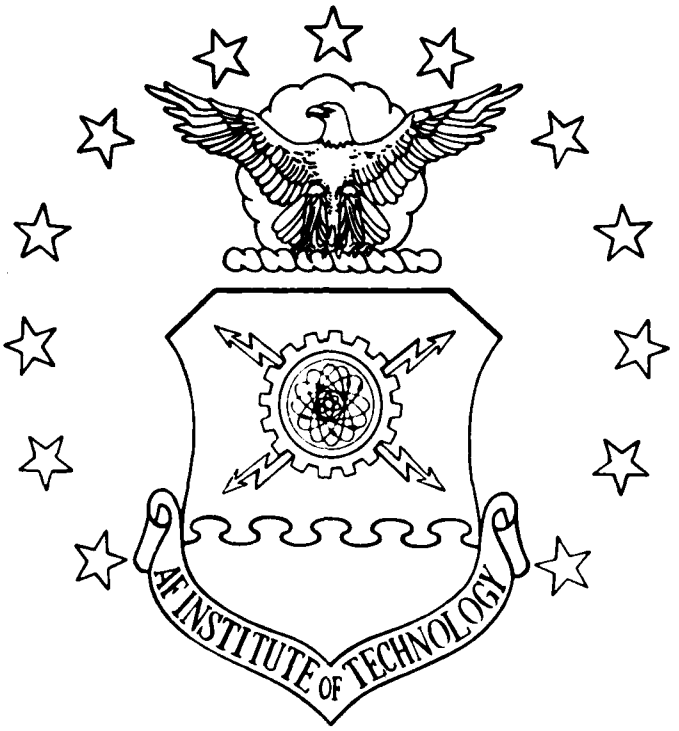
AD-A164 013 THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K H MAHONEY 1/3

UNCLASSIFIED DEC 85 AFIT/GCS/ENG/85D-7 F/G 9/2 NL







THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR FOR A DISTRIBUTED DATABASE MANAGEMENT SYSTEM

THESIS

Kevin H. Mahoney
Captain, USAF

AFIT/GCS/ENG/85D-7

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/85D-7

THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR FOR A DISTRIBUTED DATABASE MANAGEMENT SYSTEM

THESIS

Kevin H. Mahoney
Captain, USAF

AFIT/GCS/ENG/85D-7

Approved for public release; distribution unlimited

AFIT/GCS/ENG/85D-7

THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR FOR A DISTRIBUTED DATABASE MANAGEMENT SYSTEM

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree

of Master of Science in Computer Systems

Kevin H. Mahoney

Captain, USAF

December 1985

Preface

The purpose of this study was twofold. The first purpose was to analyze and define the requirements for a universal, or global, query language for use in a heterogeneous distributed database management system, and to propose query translation algorithms based on the use of the relational model as the global data model. The second task was to design and partially implement translator software for translating relational queries into data manipulation language of the TOTAL network data base management system.

I would like to acknowledge the support and encouragement that I received from my thesis advisor, Dr. Thomas Hartrum and from my reader, Dr. Henry Potoczny. I would also like to thank Dr. Gary Lamont for the use of the Information Sciences Laboratory VAX-11, and Capt Dave Gaitros for his much-needed help in learning the intricacies of the TOTAL DBMS and AFIT Data Base.

Finally, I would like to thank my wife, Mickie, and the rest of my family, for their constant understanding, support, and encouragement, without which I could not have finished this work.

Kevin H. Mahoney

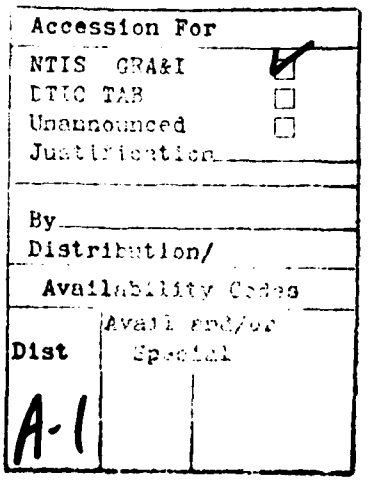


Table of Contents

List of Figures

List of Tables

Abstract

A translation program was implemented for the translation of generic relational queries into the Data Management Language (DML) of the TOTAL data base mangement system. The objectives of this thesis were to propose and detail a method of supporting a global relational query language for a heterogeneous distributed database management system (DDBMS), design query translators for the translation of global relational queries into local hierarchical and network DML, and partially implement translator software for the conversion of relational queries into TOTAL DML.

The initial portion of the thesis presents an overall analysis of data partitioning, query decomposition and global query management in the DDBMS. Specific proposals are advocated concerning the specific approach to be taken toward a global query manager and the design of the global schema over current databases.

The second portion formalizes the assumptions and constraints present in the global relational model, and presents generic algorithms for the translation of relational queries into hierarchical and network DML.

The last portion details the implementation and testing of the relational to TOTAL translator. The approach used was to "compile" the relational query into a generated Pascal program containing the TOTAL DML commands. Only the actual code generation portion of the software was implemented. Query parsing, query optimization, and results integration were not addressed. A set of sample queries translated by the program were presented and evaluated.

I. Introduction

Background

Nearly all of the database management systems (DBMSs) in use today are one of three general models -- network, hierarchical, and relational. The hierarchical Information Management System (IMS) was the first major DBMS, introduced by IBM in the 1960s. Efforts to standardize DBMSs led to the CODASYL standard, a network DBMS known as the DBTG model (after the CODASYL Data Base Task Group). This standard was introduced in the late 1960s and facilitated the standardized development of many new DBMSs using the network model. However, at about the same time, E. F. Codd proposed a DBMS based on the mathematical principle of the relation. The relational DBMS was born, and continues to be the object of most data base research and new product development to this day.

At the same time that these developments were taking place, the use of data communications, or computer networking, was growing throughout industry as a whole. The need for common information distributed across several locations (for example, account information at a bank with several branches) led to the development and use of the distributed data base management system, or DDBMS. Most of these systems were tailor-made for the user. However, several large companies and the government (especially the government) had already made major investments in DBMSs, which often times were based upon the different models (relational, network, hierarchical). A way to tie these different DBMSs together, using a network, was needed. This

problem was originally identified by Adiba (1) as the "communication and cooperation of heterogeneous databases".

The initial problem to overcome was that the DBMSs usually ran on different computers. However, with the implementation of computer networks, it became possible to tie all of these DBMSs together. Today, even with networking, the practical distributed use of the available DBMSs is still very limited. Users must still know which specific system has the information that they seek, and they must also be able to use that particular DBMS's query language in order to access the information.

The current situation at the Air Force Institute of Technology (AFIT) is a good example of this "heterogeneous distributed data base problem". AFIT has several DBMSs that it would like to tie together using a local network. The AFIT DBMSs include: TOTAL, a network DBMS, running on a VAX 11/780 computer using VMS, several different relational DBMSs, including INGRES on two VAXs running UNIX and the one running VMS, ORACLE on a Harris minicomputer, and dBase II on several CP/M-based microcomputers. Each of these systems stores or has the potential to store information needed by the faculty, staff, and students of AFIT.

Problem Statement

Several efforts have been made or are currently underway to tie these heterogeneous data bases together. However, one major area yet to be addressed is the lack of a "universal" or "global" query language that could be used to retrieve information from any of the

several different DBMSs. A single query language would make it much easier for a user to get needed information, without having to know which DBMS holds the information and what the query language is for that particular DBMS. The purpose of this thesis is twofold; to propose a method of implementing this capability for the AFIT distributed data base, and to implement a portion of the system that will translate "global" relational queries into the data manipulation language (DML) for the TOTAL DBMS.

Scope of the Effort

The research and specification will be for a read-only system, one capable of executing typical relational queries (project, select, and join). The initial need for the majority of current AFIT users is to gain access to the distributed information, not to update it.

The partial implementation will be to develop the translator to the TOTAL DBMS DML, since the AFIT Master Data Base (which is implemented using TOTAL) is the most widely used of any of the various AFIT data bases.

Assumptions

The relational model and query language will be used as the universal model and language. The reason for this choice is outlined in Jones' thesis (6). For the partial implementation of the TOTAL translator, the Roth Relational Database query language will be the universal language used. There are two reasons for this choice: First, Roth is already being used as the DBMS query language on the

LSINET. Second, translators have already been developed to convert Roth queries to dBase II and INGRES language queries.

The query translation will be one-way from the global relational language to the local DBMS query language. It is assumed that all queries will originate in the Roth language and then will be translated into the local DBMS query languages. Local DBMS queries will not be translated into the global language and then back into another local language.

Summary of Current Knowledge

Several research efforts are being made in this area, but there is a lack of recent publications. The majority of documented research appears to be for the period 1976 through 1980. The major studies conducted were at the University of California, Los Angeles (UCLA), the Computer Corporation of America, Grenoble University in France, and the Japanese Information Processing Development Center (JIPDEC). Several other research projects also have been done on various aspects of heterogeneous DDBMSs, and this thesis draws heavily upon this previous research. Most of the available literature concerns the architecture and data mappings (schema conversions) of such a system. Research into the mapping of queries does not seem to be as common.

Approach

This effort was divided into three basic stages; background research, high-level design of the query translator algorithms, and the implementation of the relational-TOTAL translation software. After the system requirements were defined and analyzed, the issues of

query decomposition and mapping between the various query languages were addressed. This was done by tracing the flow of query transactions through the distributed data base network. The final objective was to partially implement query translation software that takes the information provided by the relational query, and generates an equivalent program encompassing the required TOTAL DML statements.

Sequence of Presentation

This thesis is organized into ten chapters. Chapter Two covers the current knowledge on heterogeneous DDBMSs in depth and proposes an approach for the AFIT DDBMS. Chapter Three examines duplication and fragmentation of information throughout the DDBMS. Chapter Four addresses the global query manager and the decomposition of queries. Chapter Five defines the relational query language and restrictions that are placed on the DBMS schema. Chapter Six describes the mappings to IMS DML, and Chapter Seven describes the mappings to CODASYL DML. Chapter Eight describes the mappings to TOTAL. Chapter Nine describes the partial implementation and testing of the relational to TOTAL translator. Chapter Ten contains the final conclusions and recommendations.

II. Heterogeneous DDBMS Approach Selection

Introduction

Before proceeding with the problem analysis, the various types of heterogeneous DDBMSs that have already been proposed and/or developed should first be surveyed. The six types of DDBMS defined by Katz (10) will be used to describe the heterogeneous DDBMS alternatives, one of which will be chosen as the approach to follow. That decision is then reinforced by the following discussion of the major points of four preceding AFIT theses whose work in relational and distributed data bases formed much of the initial direction and foundation of this thesis.

Approaches Toward Heterogeneous DDBMSs

In 1981 Katz defined six different approaches towards implementing a heterogeneous DDBMS, based upon ongoing research in the field. These approaches, and their respective major attributes, are shown below in Table 1.

		integrated	
	separate	disjoint	overlapped
single global model	Database Terminal	Database Prism	Database Prism
multiple global models	Database Window	Windowed Database Prism	Windowed Database Prism

Table 1 - Types of DDBMS Systems (10:36)

Database Terminal. The Database Terminal system would be one that uses a global data model and query language, but in which the various database schemas are not integrated. This gives a common access method to different databases. One example of this type of system is proposed by Date (6:449-468).

Database Window. The Database Window is similar to the Terminal system in that the separate database schemas remain unintegrated, but differs in that several models can be used as the global model. For example, relational DML could be used to access a network database or network DML could be used to access a relational database, both on the same terminal. Zaniolo (20:189) discusses some of the issues involved, but no prototype system has yet been proposed.

Database Prism. The Database Prism differs from the above approaches in that the local database schemas are integrated into a global schema organized under a single global model. The user issues the query against this global schema in the DML of the global model. The system then maps the query into a set of queries against the local databases. The actual location of the underlying data is transparent to the user. Katz chose the term "prism" because the system splits a query into different parts much the same way that a prism splits a beam of light into several colors.

The Database Prism can be further divided into two classes. The local schemas can either be disjoint or they may overlap. If they are disjoint, the system is much simpler to implement. If they overlap, then there is the problem of dealing with possibly subtle differences

between the same data stored at different databases. The MULTIBASE and POLYPHEME systems are proposed overlapping Database Prisms.

Windowed Database Prism. The Windowed Database Prism is where the features of both the Window and Prism systems are combined. The local database schemas are integrated into a single global schema, but this global schema may be accessed through several different data models and DMLs. This system is also divided into two classes, for disjoint or overlapping schemas, with the same inherent difficulties as the Prism for overlapping data. The major research into this type of heterogeneous DDBMS is taking place at U.C.L.A. (5).

Choice of Heterogeneous DDBMS

Given the aforementioned different approaches to implementing a heterogeneous DDBMS, which one is the most appropriate one to use? There are three major criteria for the heterogeneous DDBMS proposed for AFIT:

1. Be able to access data that spans local databases without the user having to know the actual location.
2. Be able to deal with overlapping local schemas.
3. Be reconfigurable to a point that eliminates critical nodes in the DDBMS network.

These requirements most closely match the Database Prism approach, which is the method proposed by this thesis. Not surprisingly, this is also the method that previous AFIT theses dealing with distributed databases (summarized below) have been heading towards.

Imker and Boeckman Theses

The AFIT theses by Capt Eric Imker (1982) and Capt John Boeckman (1984) involved the initial design and implementation of a DDBMS for the AFIT Digital Engineering Laboratory (DEL). This work is presently being continued by Capt James Wedertz. The main features of the system initially proposed by Imker and expanded upon by Boeckman (3) are as follows:

1. The DDBMS is a reconfigurable system. Nodes can be added to or deleted from the system and the central site can be relocated from one site to another.

2. The central site maintains the Central Network Data Dictionary (CNDD) which maintains information on all data stored throughout the DDBMS.

3. Each site maintains a Local Network Data Dictionary (LNDD) that maintains information on data stored at that site.

4. Each site maintains an Extended Network Data Dictionary (ECNDD) that maintains information on some of the data stored at other sites in the system. The ECNDD is permitted to grow only to a given size, usually smaller than the size of the CNDD.

5. Status information tables are maintained on every site in order to determine if the site is active or not.

6. Each site is capable of handling queries and updates.

Jones' Thesis

The thesis by 2Lt Anthony Jones (1984) involved the analysis and specification of the universal data model and Data Definition Language (DDL) for a heterogeneous DDBMS. Jones' thesis provides the foundation of this thesis by his choice of a relational model as the global DDBMS model. **NOTE:** The use of Jones' term "universal" for the data

model and language corresponds to the term "global" in this thesis. This change was done to avoid possible confusion with the Universal Relation model described by Ullman (18:317-346). The major features of the system proposed by Jones (9) are as follows:

1. The Relational Model was chosen as the global data model for the DDBMS.

2. The DDL was specified for data translation to and from the CODASYL model (network), the IMS model (hierarchical), and the System R model (relational).

3. A Universal Data Base Administration Center (UDBAC) was proposed to handle the formation of new databases, deletions and updates of data, security issues, and other DDBMS policies. This UDBAC maintains a very powerful role in the running of the system.

4. There would be a seperate UDBAC computer with relational power that would be used to assemble intermediate results before routing the final product to the requesting node.

5. Only First Normal Form will be required for the global model, although Third Normal Form will be used as often as possible.

6. Duplicate keys (IMS and CODASYL) will not be allowed.

7. Partial replication of data must be supported, but full replication or no replication are the preferred choices.

8. Only the "manual insertion class" will be allowed for data that is stored in an underlying CODASYL database.

Roth's Thesis

The thesis by 2Lt Mark A. Roth (1979) partially implemented a relational database system on a microcomputer. His thesis was used as supporting research for this thesis in regards to the query language. The Roth DML (a relational algebra) was used as the global DML in Boeckman's partial implementation of the DDBMS. The query parsing

methods that Roth used (a forest of binary trees with pointers to attribute lists) were used as the basis for a simple global query parser for the DDBMS that was implemented by Wedertz.

Summary

A quick review of the approaches defined by Katz and the preceding AFIT thesis work indicated that the heterogeneous DDBMS to be implemented will be of the "Database Prism" type. The next three chapters of this thesis are a more detailed discussion of the requirements that the DDBMS must meet in order to achieve an effective implementation of a read-only global query language.

III. DATA PARTITIONING AND REDUNDANCY

Introduction

This chapter discusses the partitioning of global relations into "fragments" across the various data bases of the DDBMS. This is a very real concern that will most probably be present in any DDBMS that is created from existing local databases, but is also an important consideration in DDBMSs (both homogeneous and hetero- geneous) that are designed from the ground up. This chapter is not a discussion of the "best" way to partition the data -- that task is not a function of the system, and is best left to the data base administrator for the finalized DDBMS.

What is discussed in this chapter is the range of data partitioning and redundancy that the DDBMS should be capable of handling. The importance of such a capability becomes apparent when the DDBMS must be able to decompose a global query into the appropriate (and most efficient) local DBMS queries. That aspect of the partitioning problem will be covered in the next chapter.

Partitioning of Relations

As discussed in the previous chapter, the global relation that the DDBMS user sees can actually be made up of several local relations, also known as fragments. According to Ullman (18:411), relations can be partitioned (fragmented) in two ways, vertically and horizontally.

Vertical Partitioning. Vertical partitioning is when the set of attributes for each of the partitions is a subset of the global relation's attributes. It is so named because if one views a relation as a table, then the partitions would represent vertical columns of the table. The method of composing the partitions into a single relation is by using the natural join, which implies that each fragment must contain an attribute that is shared by at least one other fragment.

$$\text{Global Relation} = R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$$

Horizontal Partitioning. Horizontal partitioning is so named because the table (relation) is being partitioned by rows, or tuples, thus creating horizontal layers of tuples. These fragments each contain the same attributes. The partitions are composed into a single global relation by using the union operator.

$$\text{Global Relation} = R_1 \cup R_2 \cup \ldots \cup R_n$$

Overlapping Partitions

In both of the above classes, the partitions of the database cannot be assumed to be disjoint. That is, data in the local relations could possibly overlap each other. For the following examples, assume there is a single global relation consisting of attributes "a", "b", "c", "d", "e", and "f", with attribute "a" being the key for the relation.

Vertical Disjoint and Overlapping Partitions. If the relation is vertically partitioned into disjoint partitions (relations) R_1, R_2, and R_3, then the intersection of each relation's set of attributes (not the tuples), denoted by R_iA, would show:

$$R_1A \cap R_2A \cap R_3A = \{a\}$$
(a being the key needed to effect the join)

However, if the partitions were not disjoint, and attributes "b" and "d" overlap between them, then the intersection of the sets of attributes would show:

$$R_1A \cap R_2A \cap R_3A = \{a,b,d\}$$

Horizontal Disjoint and Overlapping Partitions. If the relation in this example was horizontally partitioned into disjoint relations R_1, R_2, and R_3, the intersection of the relations' tuples (not the attributes) would show:

$$R_1 \cap R_2 \cap R_3 = \{\} \quad \text{(the empty set)}$$

If the partitions overlap, then the intersection of the relations' tuples would show:

$$R_1 \cap R_2 \cap R_3 = \{\text{one or more tuples}\}$$
(tuples exist in more than one partition)

Overlapping Attribute Inconsistencies

Overlap of attributes in vertical partitioning could cause additional difficulties. According to Katz (10:37), when the schemas of local databases overlap, a way must be found to identify possibly

subtle differences between data stored about like objects within different databases. This incompatibility can take two forms; like data with different local attribute names, and/or actual data discrepancies such as the difference between "red" and "scarlet".

The first condition, different attribute names, must be addressed through the data dictionary for the DDBMS (not within the scope of this thesis). The data dictionary would take the global attribute name as input, and return each of the associated local attribute names to the system for further processing.

To handle the second condition, Katz (10:38) proposed the creation of a separate database known as an "integration schema", where values of attributes that are used in comparisons can be examined to determine if there are other valid data values, such as "scarlet" being the same color as "red". This method could also be used to make data conversions, such as meters to feet, degrees Celsius to degrees Fahrenheit, and so forth.

Redundancy of Data

The previous sections discussed the partitioning of data within the DDBMS. The other problem that must be addressed is how to handle the redundancy of data. When integrating several existing DBMSs into a global DDBMS, there may be a large amount of duplication, or redundancy, present in the global system. For example, a person could have a file in a personnel database, a separate file in a student database, and another file in a payroll database. Most of the information used

in each database (name, address, age, etc.) is redundant. This type of data redundancy falls into three categories:

1. No Redundancy - The data is unique within the DDBMS.
2. Full Redundancy - Each relation is fully duplicated at least at one other location, and possibly across several locations.
3. Partial Redundancy - Part of a relation's attributes and/or tuples are duplicated elsewhere, but a complete duplicate set of the data does not exist.

Full Redundancy. In the case of full redundancy, the redundant information can exist as exact duplicate relations, with the same domains and cardinality. Or the duplicate data could be a subset of another, larger, relation. Redundant data could exist across several partitions, which could also be duplicated. A relation could be both fully and partially redundant at the same time, if there exists at least two full copies, plus at least one partial copy, of the relation. The possible combinations are nearly endless.

Partial Redundancy. In partial redundancy, there are even more possible combinations. The most prevalent form of partial redundancy occurs when vertical or horizontal partitions of a global relation overlap, creating duplicate attributes or tuples, respectively, within the relation. For effective processing, the DDBMS should be capable of addressing all these types of redundancy, since they will most likely be present in some form, especially when the DDBMS incorporates existing local DBMSs.

Partitioning/Redundancy Classes

Now that the basics have been outlined, what different types of partitioning and redundancy should the DDBMS be capable of handling? The next few pages define ten partitioning/redundancy classes that the system should be designed to handle. The attributes of the relations are listed as "a", "b", "c", "d", ..."z", with "a" considered to be the key attribute (possibly a composite key) of the global relation. Separate local relations are denoted as "LRel" with a numbering subscript. Each class is accompanied by a figure showing a graphical representation of the partitioning. Note that in vertical partitioning, the duplication of the key attribute(s) is not considered redundant, since it is needed to effect a join of the partitions.

Class 1 - No Partitioning and No Redundancy. In this class, the given relation is unique. The data is stored as one relation at one site. In Figure 1, the global relation is the same as the local relation.

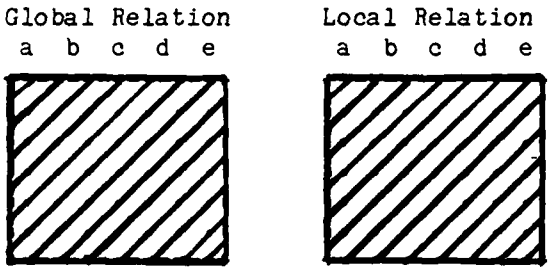


Figure 1 - Class 1 Partition

Class 2 - No Partitioning and Complete Redundancy. In this class, the global relation is composed of one local relation that is stored entirely at a single site as a complete relation. However, there is at least one complete duplicate copy of the information in the DDBMS.. The duplicate relation(s) may be stored as a single relation at one site, may be a subset of a larger relation, or it may be partitioned. In Figure 2, the global relation can be found in its entirety at three different sites, the third of which (LRel3) is a subset of a larger local relation.

$$\text{Global Relation} = \text{LRel}_1 \vee \text{LRel}_2 \vee \ldots \vee \text{LRel}_N$$

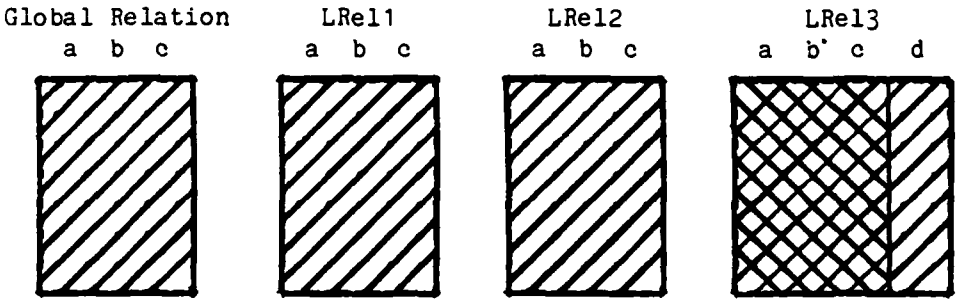


Figure 2 - Class 2 Partitions

Class 3 - Vertical Partitioning (No Redundancy). In this class, the data is not duplicated anywhere, but neither does the entire relation exist fully in one location. The global relation is composed of vertical fragments, and there must be a foreign key (or the global key) present in each of the fragments in order for the natural join to occur. For example, in Figure 3, the global relation is vertically partitioned into two local relations. The "(b)" in LRel2 signifies that the foreign key for the join is attribute "b", which is present

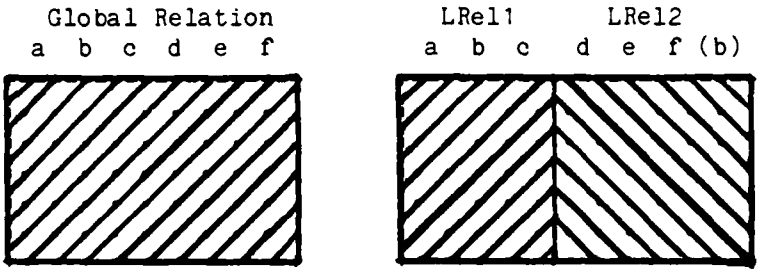


Figure 3 - Class 3 Partition

in both LRel1 and LRel2. Attribute "b" is not considered redundant, as it is needed to effect the join.

$$\text{Global Relation} = \text{LRel}_1 \bowtie \text{LRel}_2 \bowtie \ldots \bowtie \text{LRel}_N$$

Class 4 - Vertical Partitioning (Partial Redundancy). In this ss, attributes of the global relation overlap each other within the local relations. The duplicate attributes are eliminated in a natural join, but knowledge of which attributes exist in each local relation could prove useful if only part of the global relation's attributes are to be projected in a query (possibly eliminating the need for a join). In Figure 4, attributes "c" and "d" overlap, in addition to the global key attribute "a".

$$\text{Global Relation} = \text{LRel}_1 \bowtie \text{LRel}_2 \bowtie \ldots \bowtie \text{LRel}_N$$

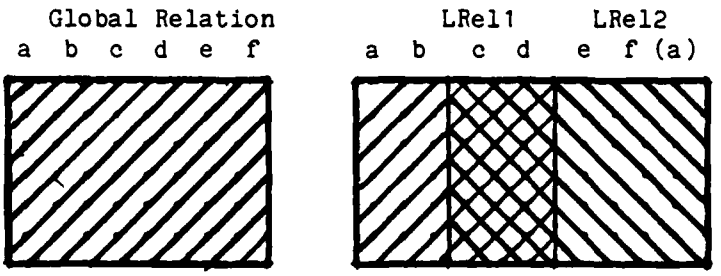


Figure 4 - Class 4 Partition

Class 5 - Horizontal Partitioning (No Redundancy). In this class, each of the local relations possesses all of the necessary attributes of the global relation, but no single one of them contains all of the tuples of the global relation. Figure 5 shows the horizontal partitioning of the global relation.

$$\text{Global Relation} = \text{LRel}_1 \cup \text{LRel}_2 \cup \ldots \cup \text{LRel}_N$$

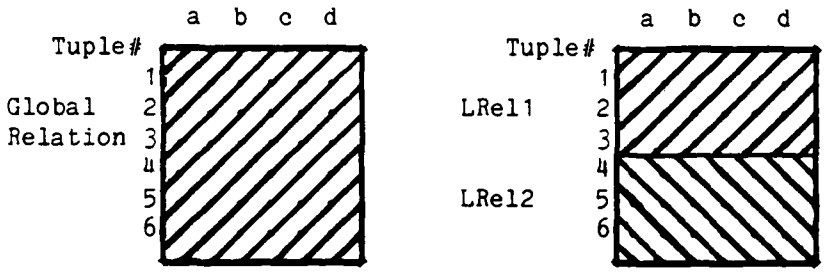


Figure 5 - Class 5 Partition

Class 6 - Horizontal Partitioning (Partial Redundancy). Here each of the local relations possesses all of the necessary attributes, and some of the tuples exist in more than one local relation. In Figure 6, tuples 4, 5, and 6 overlap.

$$\text{Global Relation} = \text{LRel}_1 \cup \text{LRel}_2 \cup \ldots \cup \text{LRel}_N$$

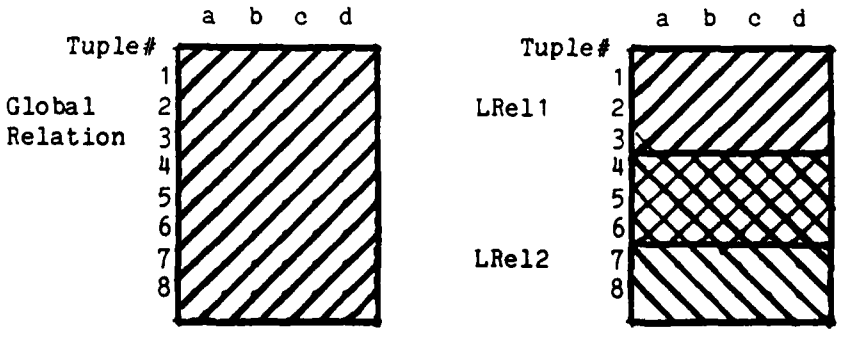


Figure 6 - Class 6 Partition

Class 7 - Vertical/Horizontal Partitioning (No Redundancy). This class is a combination of classes 3 and 5. The global relation is composed of two or more vertical partitions, one or more of which are further divided into horizontal partitions. For proper recomposition of the relation, the union of the horizontal partitions must be accomplished before the join of the vertical partitions. In Figure 7, the global relation is partitioned into three local relations, LRel1/LRel2 and LRel3 forming the vertical partitions, with LRel1 and LRel2 partitioned horizontally.

$$\text{Global Relation} = (\text{LRel}_1 \cup \text{LRel}_2) \bowtie \text{LRel}_3$$

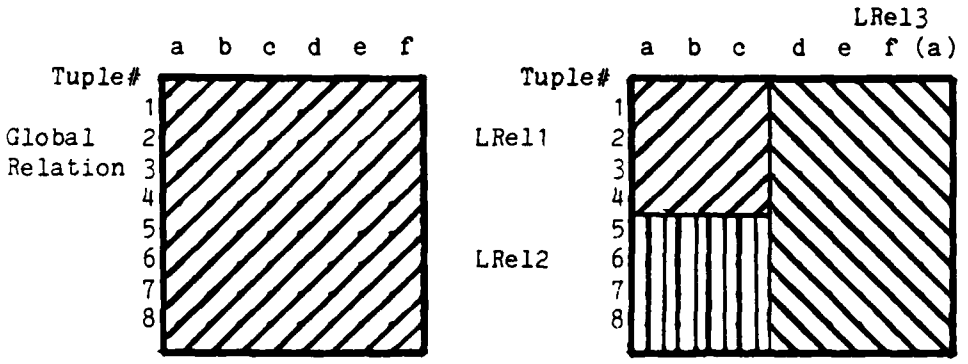


Figure 7 - Class 7 Partition

Class 8 - Vertical/Horizontal Partition (Partial Redundancy). This class is a combination of classes 4 and 6. The global relation is composed of two or more vertical partitions, one or more of which are further divided into horizontal partitions. What is different in this class is that the horizontal partitions are partially redundant, and the vertical partitions could also be partially redundant. However, processing for both is the same as for the previous class.

In Figure 8, tuples 3, 4, 5, and 6 horizontally overlap. In Figure 9, attribute "c" also overlaps vertically.

$$\text{Global Relation} = (\text{LRel}_1 \cup \text{LRel}_2) \bowtie \text{LRel}_3$$

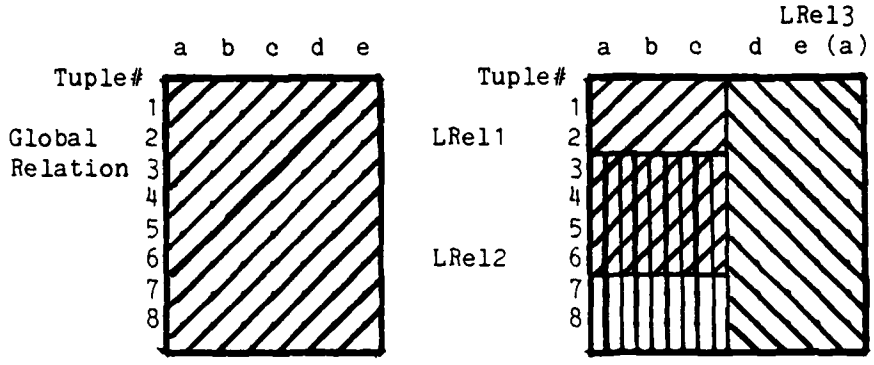


Figure 8 - Class 8 Partition
(Horizontal Overlap Only)

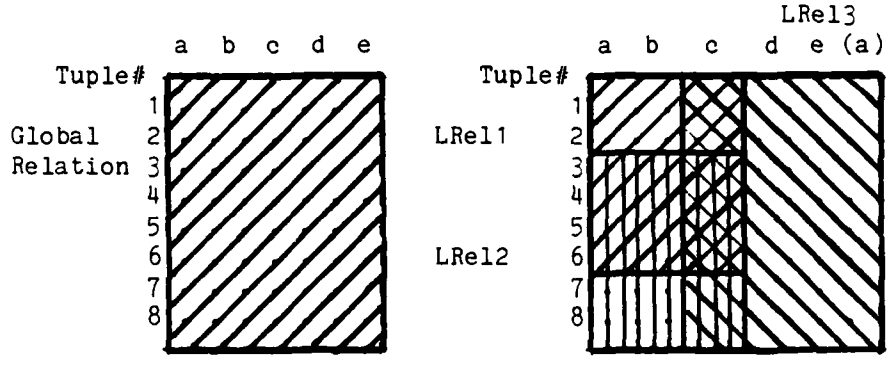


Figure 9 - Class 8 Partition
(Horizontal and Vertical Overlap)

Class 9 - Horizontal/Vertical Partitioning (No Redundancy).

This class is a combination of classes 5 and 3. The global relation is composed of two or more horizontal partitions, one or more of which are further divided into vertical partitions. For proper

recomposition of the relation, the join of the vertical partitions must be accomplished before the union of the horizontal partitions. In Figure 10, the global relation is partitioned into three local relations, LRel1/LRel2 and LRel3 forming the horizontal partitions, with LRel1 and LRel2 partitioned vertically.

$$\text{Global Relation} = (\text{LRel}_1 \bowtie \text{LRel}_2) \cup \text{LRel}_3$$

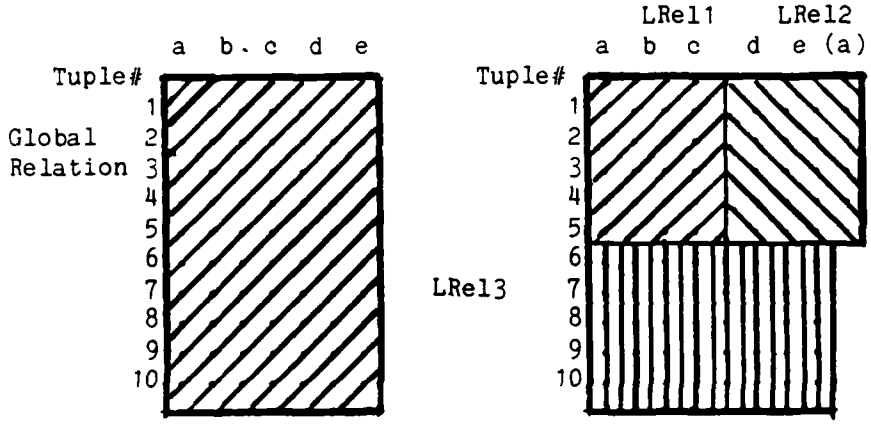


Figure 10 - Class 9 Partition

Class 10 - Horizontal/Vertical Partition (Partial Redundancy). This class is a combination of classes 6 and 4. The global relation is composed of two or more horizontal partitions, one or more of which are further divided into vertical partitions. What is different in this class is that the horizontal partitions are partially redundant, and the vertical partitions could also be partially redundant. However, processing for both is the same as for the previous class. In Figure 11, attributes "c" and "d" vertically overlap. In Figure 12, tuples 4, 5, and 6 overlap horizontally as well.

$$\text{Global Relation} = (\text{LRel}_1 \bowtie \text{LRel}_2) \cup \text{LRel}_3$$

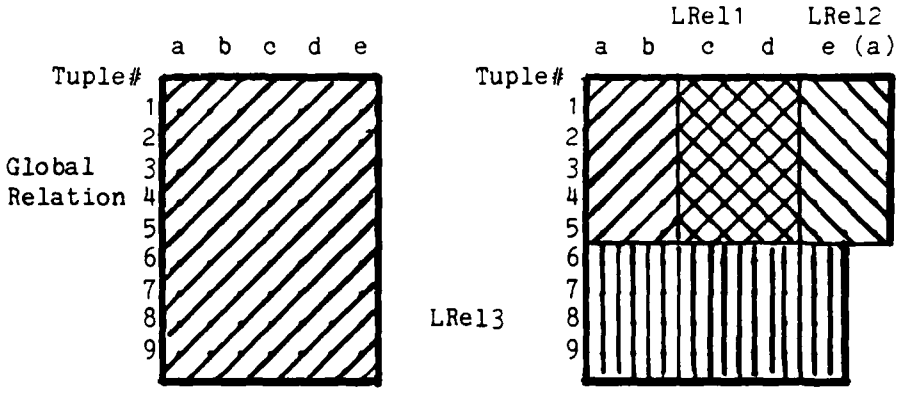


Figure 11 - Class 10 Partition
(Vertical Overlap Only)

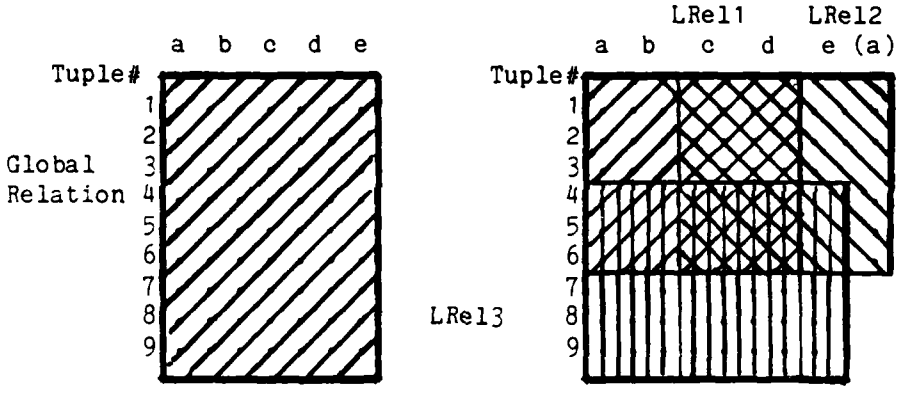


Figure 12 - Class 10 Partition
(Vertical and Horizontal Overlap)

Summary

In this chapter, the different types of data partitioning and redundancy that are possible in a distributed database built from existing DBMSs were examined. These issues are important for the decomposition and optimization of queries (and updates) across the DDBMS, which is the subject of the next chapter. Ten different classes of partitioning were defined, and the global relations of the

system that fall under these classes will need to be identified by the DDBMS data dictionary at the time of the query. Information returned by the data dictionary will be used for the decomposition of queries and recomposition of query results in the DDBMS.

IV. Global Query Management in the DDBMS

Introduction

In the previous two chapters, the different types of distributed databases and the type of data partitioning that they must handle were outlined. In this chapter, the global query manager part of the DDBMS architecture is examined. The majority of the DDBMS architecture has been previously defined by Boeckman (3) and Jones (9), and the major points of each were outlined in Chapter 2. What is presented here is a closer look at just the query handling aspects of the system. First, the general aspects of a global query manager are addressed, and then specific correlations to the AFIT DDBMS are discussed.

Global Query Manager Functions

According to Gligor and Luckenbaugh (7:34), there are five main functions of the DDBMS global query manager: (1) global data model analysis, (2) query decomposition, (3) execution plan generation, (4) query translation, and (5) results integration. These five functions are common to all DDBMSs, both heterogeneous and homogeneous, but the query translation and results integration pose special problems in the heterogeneous case. The next few sections give a brief overview of each of these functions.

Global Data Model. The global data model is the critical element of the DDBMS global query manager. Through the global schema, it provides the foundation for both the global view and the global query language presented to the system user. Key to the global data model

is the presence of a data directory, usually also distributed, which keeps track of the local schemas and information that compose the global DDBMS. The data directory also usually contains the information needed for query decomposition (such as the partitioning classes noted in Chapter 3) and for query translation into other query languages, which is the subject of the remainder of this thesis.

Query Decomposition. The query decomposer part of the global query manager takes the data partitioning information provided by the data directory and uses it to generate the set of local subqueries that retrieve the requested information. In principle, the decomposition strategy does not differ in heterogeneous systems from that of homogeneous ones (7:35).

Execution Plan Generation. The execution plan generator decides which subqueries will be sent out to the local DBMSs, which queries must precede others, and how the intermediate local subquery results will correspond to each other. This function is essentially the same for both homogeneous and heterogeneous databases, but in heterogeneous systems, the underlying local data base model may be one of the criteria used in the decision-making process. (For example, given a global relational model, if redundant data is present on both a relational and network system, the subquery would be sent to only the relational system.) For excellent in-depth discussions of this aspect of the global query manager, the papers by Hevner and Yao (8) and Bernstein, et al, (2) should be consulted.

Query Translation. It is assumed that the DDBMS user submits the query in the global DDBMS language, in this case, relational. If, as in this case, the underlying system is heterogeneous, query translation will be needed in at least two areas. After execution plan generation and decomposition of the global query, each of the subqueries will need to be translated into the local DBMS query language. After execution of the subquery, the results must be presented in, or translated into, a format suitable for the global result integrator.

Results Integration. The result integrator, or recomposer, takes the intermediate results generated by the subqueries and combines them into a global schema that is presented to the user. Once again, this process is dependent upon information that is provided by the system data directory.

Query Management in the AFIT DDBMS

The preceding sections defined the five functions that the global query manager must perform in a DDBMS. The following sections first outline those functions as they have been defined by previous AFIT thesis work, and then propose (if needed) specific recommendations for implementing these functions within the AFIT DDBMS. If the recommendation of this thesis differs from preceding works, it will be noted. Figure 13 illustrates the various functions (and the levels at which they are viewed) of the global query manager as proposed for the AFIT DDBMS.

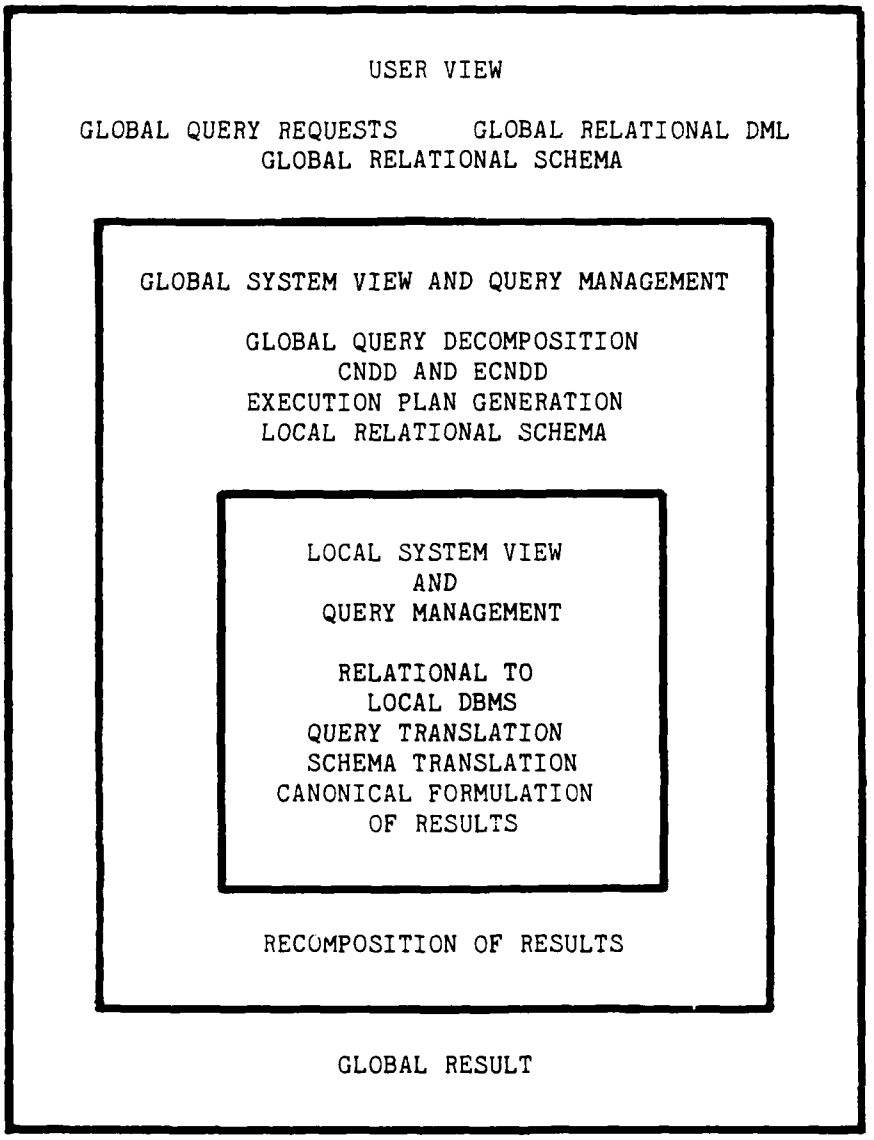


Figure 13 - DDBMS Query Management Functions and Levels

Global Data Model

The global data model for the DDBMS was defined as the relational data model by Jones (9). The initial query language to be used by the system is the Roth Relational Query Language (15). However, the data directory portion of the system was defined by Boeckman in his thesis

(3). The data directory for the AFIT system does not exist in one single location (as proposed by several other works), but exists in three separate levels. There is a Central Network Data Directory (CNDD) for the system, but each node in the DDBMS also contains a copy of a portion of the CNDD, known as the Extended Central Network Data Directory (ECNDD), as well as the Local Network Data Directory (LNDD) which contains the bulk of the information about the local DDBMS. In this case, the CNDD and ECNDD contain the information needed for query decomposition and the LNDD contains the information needed for query translation into other query languages. Each node in the DDBMS network contains an ECNDD and LNDD, but only one node will contain the CNDD. However, the host node for the CNDD is not fixed, since another feature of the DDBMS proposed by Boeckman is that it is to be reconfigurable.

This thesis follows the current system approach that specific information for the translation of the queries into the local DBMS query language should be reserved for the LNDDs. The global view should be that the global relational schema is composed of local relational schemas. However, some knowledge of the underlying local DBMS (network, hierarchical, or relational) at the ECNDD and CNDD would be beneficial for efficient decomposition of queries in cases where data is redundant. If the same information is present at both a network and relational database, it would be more efficient to decompose the query in a manner such that the subquery goes to the relational DBMS with the information.

Query Decomposition

The query decomposer portion of the AFIT DDBMS is a three step process (3:30-36). The query first consults the LNDD to see if it is a completely local query. If not, then the ECNDD is consulted. If all of the needed information is available from the ECNDD, then the query is decomposed at that point. If the necessary information is still not available, then a request is sent to the CNDD. The CNDD returns the location and other useful information (such as the type of underlying data model) to the requesting node. The query is then decomposed at the node where it orginated. The main point to note here is that query decomposition must be able to be done at each of the nodes in the DDBMS. This will minimize message traffic in the system, but will require a copy of the query decomposer at each of the nodes of the network.

Since the global model of the AFIT DDBMS is relational, it is recommended that the decomposition of the queries also follow the relational format. The global relational query will be decomposed into a set of relational subqueries against the local databases, which are viewed by the system as relational schemas. However, as noted above, knowledge of the underlying DBMS may be beneficial for the decomposition of queries involving redundant information.

Execution Plan Generation

No specific execution plan generator has been proposed by the previous works. A concurrent thesis effort by Capt James Wedertz (20) involved continuing Boeckman's work on the data directories for the

system, which, as noted above, are key to the generation of efficient execution plans. Therefore, the only specific recommendations are: (1) that the data partitioning classes defined in the previous chapter be made part of the CNDD and ECNDD, and (2) that the type of underlying database model (relational, network, hierarchical) be made part of the information stored at all three directories. These two recommendations will provide information that can aid in the subquery decision process.

Query Translation

This is the specific area of query management addressed by this thesis. Both Boeckman and Jones defined the global query language to be relational, Boeckman using the Roth language, and Jones proposing a language similar to QUEST. Query translation was assumed to be early in the query transaction process, resulting in early "command binding" (13:87).

The approach taken by this thesis is that translation from the global relational query language to the local DBMS language will be done using the **mapping** approach, generating a procedural query that will produce the same result as the relational query. Jones proposed the use of a **composite** language, one that uses both relational and procedural commands, but since the user's view of the global schema is a relational one, the commands that they use should also be purely relational. Secondly, the translation process should take place late in the query transaction process, resulting in late command binding. This will entail a unique copy of translation software to be located

at each DBMS site, but will insulate the global query management process from changes in the local databases. This is because the query decomposer generates subqueries against local relational schemas, regardless of the underlying local database. The relational subquery is then translated into the appropriate language at the DBMS site.

Results Integration

Jones did not discuss how local query results would be translated into a relational format, but did propose that the DDBMS use one node (with a local relational DBMS) for recomposition of the subquery results. However, in light of the reconfigurable system developed by Boeckman, this does not seem to be either feasible or the best approach. The recommendation and approach taken by this thesis is to require the query translators to produce results in a **canonical** format (7:42). Each separate translator will return the results to the requesting node in the form of a relation. The advantage of this is that for "n" local DBMSs, only 2n translators are needed, and the addition of more systems to the DDBMS would not affect the current local DBMSs. In this case, each of the nodes will have query translation and results integration software.

Summary

In this chapter, the overall picture of query management in the DDBMS was examined. The five query management functions of a DDBMS (global data model, query decomposition, query translation, execution plan generation, and result integration) were defined, and previous

AFIT thesis work in distributed database systems was examined. Specific proposals and their advantages for query management in the DDBMS were then given. The remainder of this thesis will now deal with the third aspect of query management, the translation of local queries from the global relational language into the local DBMS language.

V. Global Schema and Query Operations

Introduction

This chapter examines the global schema and language that are to be used for the AFIT DDBMS and their relationships to the underlying local schemas. The first part of the chapter presents conclusions about the global relational schema and the underlying hierarchical and network schemas that were reached by Jones. These constraints were imposed on the schemas in order to effect proper conversion of schemas and translation of queries. The next part of the chapter presents a more formal definition of some of these constraints. The final portion of the chapter addresses the global query language to be used by the system.

Jones' Global Relational Model

The relational model and language were chosen by Jones as the preferred data model for a distributed data base. In his analysis, Jones stated that the global relational model would have to satisfy certain constraints or would have to be extended in order to successfully represent underlying network or hierarchical schemas. These findings are outlined in the following sections.

Normal Forms. Jones contended that only first normal form can be guaranteed in the DDBMS (9:99). This is caused for two reasons, (1) CODASYL has different retention classes for information and (2) hierarchical systems do not allow segments (relations) to exist unless they are associated in a hierarchy to another segment.

Keys. Jones states that the primary keys for the underlying segments and records will be known to the LNDD as an aid to query processing. Duplicate keys will not be allowed.

Duplication of Information. Jones stated that the schema mapping routines result in global relations that do not eliminate all of the data redundancy present in the underlying schema, especially in the hierarchical case (9:101). However, there is a benefit gained in that some ambiguity in queries is removed.

Distributed Information. Allowing only none or fully redundant data is recommended, but he admits that provisions must be made for handling partially redundant data. This is in order to keep from having to modify the local databases that compose the global system.

Network Contstraints. Duplicate keys are not allowed (as noted above). Network retention classes are handled by augmenting the relational schema to make the class known to the user. This allows user policing of the retention, but requires some user knowledge of the underlying DBMS. The insertion class problem is solved by not allowing the Automatic retention class.

Hierarchical Constraints. Duplicate keys are not allowed (as noted above), and the key of each segment will be made known to the LNDD.

Formalization of Jones' Constraints

The following sections formalize the key constraints on the global and underlying schemas and their associated sets of operations that were originally presented by Jones. Before formally defining these constraints, definitions of schema and operation equivalence are presented.

Schema Equivalence. A database is **schema-equivalent** to another database if there exists a mapping that maps the schema S_2 of the second database to the schema S_1 of the first database such that all constraints in S_2, that are essential in the context of the first database, can be preserved in S_1 (19:89). One exception to "essential" would be that set ordering in a network database has no similar property in the relational database. Schema conversion is defined by inductive rules (9:99-134) that are applied to the structures. The resulting schema correspondences are depicted in Table 2.

Elements of Global Relational Model	Corresponding CODASYL Elements	Corresponding IMS Elements
Domain	Occurrence of	Occurrence of
Attribute	Item Name	Field Name
Relation	Record-Type	Segment-Type
Foreign Key	Set-Type Link-Record-Type	Hierarchical Path

Table 2 - Data Model Correspondence

Operation Equivalence. If databases are schema-equivalent, and each operation on the first database can be mapped into a set of operations on the second database without loss of consistency, then the databases can be said to be **operation-equivalent.** That is the task of the remainder of this thesis. In other words, given a set of schema-equivalent databases, provide the ability to map the relational operations project, select, and join to the equivalent hierarchical and network operations.

Relational Schema Constraints. A more formal definition of Jones' requirement that all underlying entity (segment, record) keys must be known to the global relation is known as the **foreign-key constraint** (21:186). This constraint states that if a candidate key X of relation R_1 is also an attribute combination of relation R_2, then every x-value that appears in R_1 must also appear in R_2. This constraint is described more formally by the notation below (19:90), which is also used in the following hierarchical operation mapping chapter.

FC_{ij} := **VALUE** $FK(R_i)$ in R_j dependent on **VALUE** $K(R_i)$

The term $FK(R_i)$ means that the key of R_i is a foreign key in R_j. The term $K(R_i)$ refers to the key of R_i. This constraint FC_{ij} means that **VALUES** $FK(R_i)$ in $R_j \subseteq$ **VALUES** $K(R_i)$. There are other consequences of this constraint that deal with the insertion and deletion of tuples. However, these consequences do not affect query operations, and so are not discussed here.

It is important to note that the foreign-key constraint does not allow for null values in FK(R_i). However, this is consistent with Jones' requirement that there be no null keys within the underlying local schema (9:92).

Hierarchical Constraints. No new constraints are placed upon the hierarchical schema, but a more formal definition of Jones' requirement of making the hierarchical keys known to the global relation is presented.

First, all fields in the hierarchical schema must be named uniquely. Second, each segment type must contain a hierarchical key (this is consistent with IMS). Third, the overlying local and global relations generated from a hierarchical segment must contain the hierarchical keys of all the ancestor segments for that segment. These hierarchical keys propagated into the relation can be thought of as foreign keys of the relation. This restriction can be shown more formally as follows (19:90):

Let 'S' be a hierarchical schema with 'k' segment types and 'm' hierarchical links. The schema mapping is:

1. For each root segment type H_i define a relation R_i such that

   a. R_i contains one attribute for each field of H_i:
   b. the key of R_i is equal to the hierarchical key of H_i.

2. For each dependent segment type H_j, for which a relation R_i has been generated for its parent segment type H_i, and the hierarchical link in which it is a child, recursively define a relation R_j such that

a. R_i contains one attribute for each field of H_j, and the attributes of the key of R_i;
b. the key of R_j is equal to the hierarchical key of H_j plus the key of R_i;
c. the constraint FC_{ij} is introduced.

Network Constraints. The foreign-key constraint FC_{ij} and its implicit consequences also apply to the the network transformations. However, unlike the hierarchical transforms, values for the foreign-keys do not necessarily need to be specified in the network case. Therefore, a new constraint NFC_{ij}, which allows for null values in the foreign-key, may apply. However, this constraint is only considered when inserting new values. Since this thesis deals only with queries and not updates, this constraint is not discussed.

The Global Relational Query Language

Design Decision Number 5 in Jones' thesis (9:92) specified that the global language would be a relationally based query language, but would not necessarily be any presently designed langugage. Boeckman used the Roth query language, one based upon relational algebra, as the language for his partial implementation of the DDBMS and for query translators to QUEL and dBase II. The follow-on work being concurrently done by Wedertz also uses Roth as the global language. As such, example translations in the following chapters are based upon Roth queries, and a description of the RETRIEVE portion of the Roth language is included in Appendix C. However, the approach taken by this thesis will be one consistent with Jones. No specific query language is required by the query translation algorithms presented in the remainder of this thesis.

The reason for not requiring a particular language is twofold. First, basing the query translator on the generic operations project, select, and join allows the potential use of any relational algebra or calculus based query language. Second, there is a growing movement within the government to make SQL the standard query language. Not tying the translation software to a specific language will make the adaption to SQL, or any other language, much easier. As such, the translation software will be developed to handle the type of input that would normally be expected to be returned from an LNDD in the DDBMS, namely the local database name and schema information.

Summary

In this chapter, Jones' constraints on the global relational schema and the underlying hierarchical and network schemas were first examined and then formally defined. Finally, the global query language issue was addressed. The next chapter will examine the translation of relational commands into hierarchical data manipulation language (DML).

VI. Translation to Hierarchical DML

Introduction

This chapter details the algorithms that are used to map the relational operations into the corresponding hierarchical DBMS data manipulation language. These algorithms were originally proposed by Vassiliou and Lochovsky (19). The target system language is based on IMS, with GET-NEXT and GET-NEXT-WITHIN-PARENT as the basic commands, with recursive ability assumed for the programming language and system. IMS DML was chosen because IMS is the most prevalent hierarchical DBMS in use. The relational operations to be mapped (project, select, join) are the ones associated with queries only.

Projection

The projection of attributes of a logical relation based on an underlying segment type requires a recursive algorithm, shown in Figure 14, coupled with a sequential search of the database. This is a direct result of generating the relation by the propagation of hierarchical keys, which means that the relation is composed of attributes drawn from more than one segment type. When these logical relations are constructed, the different segment types are necessarily placed in the same hierarchical path in the order $H_1, H_2, \ldots, H_k$, with H_1 being the highest level segment in the path. Since the segments are stored as levels within a tree, the retrieval path for all referenced segments is the same as a preorder traversal of the database, starting at H_1 and ending with H_k, the lowest level referenced in the hierarchy.

```
while (segment exists) loop
   get-next H1 segment
   if (no more segments) then
       exit loop
   output the referenced field values
   call RCU(1,k)
endwhile

recursive procedure RCU(i,k)
   i := i + 1
   if (i<=k) then
       loop
            get-next-within-parent Hi segment
            if (no more children) then
                 exit loop
                 output the referenced field values
            call RCU(i,k)
       endloop
  end RCU
```

Figure 14 - Projection Translation Algorithm

Selection

In selection, tuples of a relation are retrieved according to a qualification condition which is composed of a Boolean (**AND/OR**) of simple conditions. It is assumed that the qualification can be split into separate terms (t_i) where each term applies to only one segment type (H_i). There are three seperate cases of selections.

First Case. The first case of a selection is when the Boolean operator between terms is only **AND**. The terms are a series of simple conditions on respective segment types. It is possible that there is no qualification for a particular segment in the series, which in that case that term would be given the value **TRUE**. Such an algorithm is shown in Figure 15.

```
while (qualifying segment exists) loop
   get-next H_i segment where t_i
   if (no qualifying segment) then
       exit loop
   else
       call RCQ(i,k)
endwhile

recursive procedure RCQ(i,k)
   i := i + 1
   if (i<=k) then
       loop
            get-next-within-parent H_i where t_i
            if (no more qualifying children) then
                 exit loop
            if (i=k) then
                 output the referenced field values
            call RCQ(i,k)
       endloop
  end RCQ
```

Figure 15 - Selection Translation Algorithm (First Case)

Second Case. This occurs when the Boolean operator between terms is only **OR**. It is necessary to check the term for each H_i until a **TRUE** condition is found. When it is found, all following t_i can be disregarded. Once again, it is possible that there is no qualification for a particular segment in the series, but in this case the term is assigned **FALSE**. This algorithm is shown in Figure 16.

Third Case. The final case of selection occurs when the Booleans between terms are a mix of **ANDs** and **ORs**. In this situation the query must be converted into a normal form (conjunctive/ disjunctive). Once converted, each series of terms is processed independently using the first two algorithms. The results of these evaluations are then merged to give the final result. This type of selection query is not

```
while (segment exists) loop
   get-next H1 segment
   if (no more segments) then
       exit loop
   else if (t1) then
           call RCU(1,k)
        else
           call RETRIEVE-CHILDREN(1,k)
endwhile

recursive procedure RETRIEVE-CHILDREN(i,k)
   i := i + 1
   if (i<=k) then
       loop
            get-next-within-parent Hi segment
            if (no more children) then
                exit loop
            if (t1) then
               call RCU(1,k)
            else
               call RETRIEVE-CHILDREN(1,k)
       endloop
end RETRIEVE-CHILDREN
```

Figure 16 - Selection Translation Algorithm (Second Case)

often asked, which is fortunate, for generating a result would entail several passes over the database.

Join

The algorithms that follow are for translating a natural join of two relations. There are a large number of different cases for joins, so the given algorithms are somewhat simplified to minimize these differences. Two restrictions are: (1) that the Booleans between terms are restricted to **ANDs** (since an **OR** would require a sequential search), and (2) that the result relation contains domains from both of the joined relations.

```
while (segment exists) loop
   get-next H_i segment
   if (no more segments) then
       exit loop
   loop
       get-next-within-parent H_k segment
         where (F_k1=VALUE(F_i1)) AND ...
                  ...AND (F_km=VALUE(F_im))
       if (no more children) then
            exit loop
       output referenced values
   endloop
endwhile
```

Figure 17 - Join Translation Algorithm 1

Joins fall into two basic categories: (1) where both of the segments to be joined are located in the same branch of the Database Description (DBD) tree, and (2) where they are located in different branches. There are also variations within each category, the most important of which will be addressed.

Category 1 Joins. In this category, both segments (call them H_i and H_k) appear in the same branch. In the path $H_1,...H_i, ...,H_k$, with H_1 being level 0, $0 \leq level(H_i) < level(H_k)$. Translating this to the logical relations gives: $K(R_i) \subseteq K(R_k)$. This category can be further subdivided into several cases. Algorithms are presented for only the extreme cases, but a discussion of others is included.

The extreme cases have to do with what is called a **key-join-term.** This is the join of the key of the first relation with its equivalent part in the second. This may or may not be included in the query. If it is included, the task is made easier, since the hierarchical database is organized for this particular access. If it is not included,

```
while (segment exists) loop
   get-next H_i segment
   if (no more segments) then
       exit loop
   reset currency-pointer to start
   loop
       get-next H_k segment
         where (F_k1=VALUE(F_i1)) AND ...
                 ...AND (F_km=VALUE(F_im))
       if (no qualifying segment) then
            exit loop
       output referenced values
   endloop
   reset currency-pointer to last H_i segment
endwhile
```

Figure 18 - Join Translation Algorithm 2

then each level of the hierarchy must be sequentially searched for possible matching to each segment from the previous level. This is extremely inefficient, so a logical restriction to be placed on joins would be to require the key-join-term to be included in the query.

When the key-join-term is included, then all descendants of a segment will qualify according to that term. This is due to the uniqueness of the hierarchical keys. This means that all of the segments with the key-join-term can be retrieved sequentially and then be joined with all of their descendants according to any additional non-key-join terms. In the algorithm shown in Figure 17, **m** signifies the number of additional non-key-join terms.

In the second algorithm, shown in Figure 18, the key-join-term is not included as part of the query. Without this term, each H_i segment must be sequentially retrieved in order to be joined with all of the H_k segments, a very expensive process.

These algorithms are for the extreme ends of the spectrum of Category 1 joins. One in-between case is that of a join of segments with the same hierarchical root key. This means that only segments within the same record are joined. Another case is when the segments appear in different DBDs. However, the second algorithm can handle this particular case. In fact, the two presented algorithms should meet most join situations with only slight modifications.

Category 2 Joins. In this category, the segments H_i and H_k appear in different branches of the same DBD. Let paths $H_1,...H_c$, ...,H_i, and $H_1,...H_c$, ...,H_k, be the two branches. The hierarchical keys of H_1 through H_c are common attributes for both of the logical relations R_i and R_k, since the split into different branches of the DBD does not

```
while (segment exists) loop
   get-next H_c segment
   if (no more segments) then
       exit loop
   loop
       get-next-within-parent H_i segment
       if (no more children) then
            exit loop
       loop
            get-next-within-parent H_k segment
               where (F_k1=VALUE(F_i1)) AND ...
                   ...AND (F_km=VALUE(F_im))
            if (no more children qualify) then
                 exit loop
            output referenced values
       endloop
       reset currency-pointer to last H_i segment
   endloop
endwhile
```

Figure 19 - Join Translation Algorithm 3

begin until H_c. These common attributes are called **common-join-terms** when they are used as qualifications in a query, and are used to differentiate the cases of Category 2 joins.

The first case is when a common-join-term does not appear in the query. In this case, the join must be handled the same as the case of the same branch without key-join-terms. This means that Algorithm 2 will be used.

The second case is when the common-join-term is included. In this case, the join need only be applied to the descendants of the H_c segment, since the other terms are already common. H_c can be easily determined by working back up the branches from H_i and H_k until a common ancestor is found. Algorithm 3, shown in Figure 19, is to be used in this case. It is a generalization of several different possibilities, all dependent on the number of common-join-terms that are present as qualifications.

Summary

This chapter presented a set of generic algorithms for the translation of the select, project, and join relational algebra statements into IMS procedures. These algorithms are neither complete nor optimal, but give a good start to dealing with the hierarchical translation problem. The next chapter deals in greater depth with the associated problem of query translation to the network model data manipulation language.

VII. Relational to Network (CODASYL) DML Translation

Introduction

This chapter deals with the translation of a relational query into network DML. The network database model chosen for the translation is the network proposal of the CODASYL Data Base Task Group (DBTG). The DML syntax used in these translations is consistent with that described by Date (6:425-446).

The process of translating a query into CODASYL DML is a more complicated proposition than it was for the hierarchical (IMS) model. The main difference occurs in the choice of access paths that are available for a given query, and how the proper selection of these paths influences the DML code that is generated as a result. In IMS, the processing sequence consists essentially of selecting the starting segment that forms the root of the hierarchy and then traversing the tree. This amounts to a one-way traversal downward in the hierarchy. In the CODASYL model, however, processing can go either up or down from the starting record, and many different paths can be selected. Since this ability is so important, this chapter first deals extensively with the process of selecting the best types of access. The remainder of the chapter is then devoted to the query translation algorithm and a sample translation of a query into the equivalent DML statements, using a sample database that was first presented in Jones' thesis.

Query Efficiency

The tactics used for optimizing a relational query are significantly different if the interface is to the procedural DML of another data model. The usual relational process is to perform selections and projections first and then do joins over the reduced relations. However, this processing strategy fails in the CODASYL model because (11:428):

1. It is not possible to create temporary schema objects, such as those created as a result of a selection or projection.
2. Some joins are more efficient because they are prestored as CODASYL sets, and so should be processed first, rather than doing the selection or projection first.

The strategy for translating relational queries into the CODASYL equivalent DML then becomes the minimization of navigation required in the database.

Access Path Generation and Selection

Most of the current papers on the subject divide the task of translating a relational query into network DML into three separate tasks: generation of all access paths, selection of best access path, and "compilation" of the query into the appropriate DML. In most of these cases, the relational query language is relational calculus based, such as QUEL (11,16), or graphical, such as QBE (12), where select, project, join operators are not specified, leaving the choice of specific access paths to the DBMS. In the case of this thesis, the query language used by the DDBMS (Roth) is one based on relational

algebra, which means that these operations are specified in the query. However, the sequence of relational operations given in the relational algebra query may not be efficient, so all access paths must be examined for relational algebra languages as well.

Starting Record Selection. The first task in selecting the most efficient access path is to analyze the characteristics of all the records that are accessed by the query with the objective of making the translated DML as efficient as possible. Of critical importance is in this regard is the choice of the starting record type. There are three major reasons for this importance (12:91):

1. The starting record type is the one that creates the outer loop of the DML code that must be generated for the translated query.
2. The objective of the translation is to minimize the total number of records of this type that have to be retrieved.
3. Having the starting record type closest to the root of the query path helps to minimize the number of nested loops that have to be generated for the query.

What factors determine the choice of starting record and the sequence of following records? In the case of selections and projections, the query specifies the record type to be accessed, but in the case of a join, there is a choice of starting record types. The determining factor in this case is the access path characteristic of the record.

Access Path Characteristics. The following characteristics were enumerated by Katz (11:430) in his paper. Only the first three

characteristics are important to this thesis, because they deal with the logical navigation through the database, directly influencing the sequence of DML code generated to process the query. The remaining characteristics deal with the actual physical placement of data in secondary storage, which only becomes important when dealing with certain optimization techniques that are not within the scope of this thesis.

1. A path has an **exhaustive scan** characteristic if a retrieval of a record requires every record in that record type to be accessed. This is the most expensive access.

2. A path has an **evaluated** characteristic if it can navigate from the owner to the associated record via an owner pointer.

3. A set has **indexed** access if it can navigate from one record to another via a CODASYL set. One example would be all employees belonging to a certain department.

4. A set has **close proximity** if the entire set can be accessed as a minimal cost unit. This deals with the physical location of data in the system.

5. A path is **clustered** if all of the records are closely placed, reducing the number of physical accesses that must be made for retrieval of all records.

6. A path is **well placed** if both the parent and child records are physically closely placed. This would normally occur with the CODASYL option of "storage via set".

Intersection-Free Processing Orders

The final product of the selection process is an intersection-free access order for the query. This means that a record type is accessed from only one other previously visited record type. For example, say a query requires access to three record types: Employee,

Department, and Project. Links exist from Employee to both of the other record types. One example of an intersection-free query would use Department as the starting record type. If an Employee record was accessed through Department, then that record could not be accessed through the Project record. These intersection-free queries fall under two general classes, path queries and tree queries.

Path Queries. A path query is one where all clauses are: (1) two-variable (at most) that are based on an equality comparison (an equijoin), (2) supported by the underlying schema (CODASYL sets exist between the joined records), and (3) the query is acyclic, with no node connected to more than two other nodes. There are 2^{N-1} different orderings of a path query with N nodes. It is extremely fortunate that most queries involve few nodes, or otherwise the selection of an order would be a monumental task in itself. For an effective algorithm that generates all possible orderings of a path query, the reader is referred to (11:440).

Tree Queries. The other class, the tree query, shares the first two characteristics of the path query, but differs in that the acyclic graph is connected, which means that a node in a query with N nodes can be connected to up to N-1 nodes. The problem of generating all the possible orderings for this type of query falls under the class of NP-complete problems, which means there currently exists no efficient algorithm. Selection of a query ordering would most likely depend on a heuristic approach, one that is not within the scope of this thesis.

```
Unselected-List := query record types 1,2,...,N
Ordered-List := null

LOOP UNTIL (Unselected-List is empty)

  IF (query tree has record type with CALC key) AND
      (query provides values for CALC key attributes)
     THEN
        IF (query gives value for unique record) THEN
           Pick that record as starting type
        ELSE  {the query has several candidate records}
           Pick the record closest to the root
     ENDIF
  ELSE  {no CALC key is available}
     IF (tree has a record type that is a member of
          a system-owned set)
        THEN
           IF (search/sort key is available)
              Pick that record type
           ELSE  {no key is available}
              Sequential search required
           ENDIF
     ELSE  {not a system-owned set, either}
        Add a link to the system-owned set closest
          to the tree's root
     ENDIF
  ENDIF

  ADD selected record type to Ordered-List
  REMOVE selected record type from Unselected-List
ENDLOOP
```

Figure 20 - Order Selection Algorithm

Processing Selection Algorithm

The algorithm of Figure 20 selects the most efficient access order for a path query. It is derived from the separate works of Katz (11) and Kuck (12). It does not generate all possible access sequences, but iteratively checks all record types for the type of

access possible for that record type and then selects the most efficient access. Braces ({}) signify comments.

Ordered Record List. The selection algorithm produces an ordered list of records to be accessed, with the corresponding attributes and access characteristics. This is not unlike the Iterative Query Language, or IQL, that was proposed by Katz (11:434). Katz's IQL consists of nested FOR EACH statements, each of which is associated with a given record type. For example, the following Roth query gives the department location for the department in which John Doe works.

```
SELECT ALL FROM employee WHERE (name = 'John Doe')
     GIVING temp1
JOIN temp1, department WHERE (works-in = dept#)
     GIVING temp2
PROJECT temp2 OVER dept-location
     GIVING temp3
```

The equivalent IQL statements would be:

```
FOR EACH employee record WHERE emp.name = 'John Doe' DO
   FOR EACH department record WHERE emp.works-in =
             dept.dept# DO
      PRINT dept.location
```

The most efficient processing, in this case, would be if a CALC key exists for the "employee" record type and a CODASYL set associated with "works-in" links the "employee" and "department" record types. The worst case would be if both record types have to be searched exhaustively to find the particular record that is sought. There are also many different possible combinations in between. Whatever the access characteristic, it will be noted along with its associated record type. This modified "IQL" would take the following form:

```
FOR EACH record-type-name
  WITH access-type (USING key-attribute)
    WHERE condition DO
```

The WITH statement gives the access type (CALC, sequential search, etc.). For an access type of CALC or indexed, the USING statement gives the key attribute(s), since it might not be the same attribute that was part of the selection condition. In any case, this enumeration of the required records is what is used to generate the appropriate DML statements that will be used to process the query.

DML Generation Algorithm

Once the most efficient network access sequence has been determined, the task is to generate the necessary DML code for that particular access sequence. Generation of code was chosen over the use of set routines because of the flexibility that it offers in handling different combinations of selects, projects, and joins.

The DML generation algorithm shown in Figure 21 is based upon one proposed by Katz (11:443). This recursive descent algorithm takes the ordered list of record types and qualifiers and generates the DML statements that create the equivalent CODASYL operations for a given relational query. This algorithm handles the SELECT, PROJECT, and JOIN operators all the same way, since the ordered list is what determines which code is generated. The algorithm takes the first record and builds the outermost loop of the DML program first, and then works recursively towards the innermost loop, which processes the last record in the ordered list. LABEL(index) is a procedure that

creates labels by appending the index number to the beginning letter "L", such as L1, L2, and so forth.

```
procedure MAIN(ordered list)
  Number FOR EACH records in ordered list as 1,2,...,N
  DMLGEN(1)
end MAIN

procedure DMLGEN(i)
  IF (i > N) THEN return {no more record types left}

  IF (one-variable equality clause) THEN

     IF (access by identifier) THEN
        <key> := key name of identifier data item
        create DML string
           "MOVE <value> to <R_i.identifier data item>
                 IN <R_i>
            FIND ANY <R_i> USING <key>
            IF NOTFOUND THEN GO TO LABEL(3*(i-1))
            GET <R_i>
            IF NOT(boolean conditionals) THEN
                 GO TO LABEL(3*(i-1))"
        DMLGEN(i+1)
        create DML string
           "LABEL(3*(i-1)): "

     ELSE IF (access by indexed path) THEN
        <key> := keyname of indexed path
        create DML string
           "    MOVE <value> to <R_i.value data item>
                FIND ANY <R_i> USING <key>
            LABEL(3*(i-1)+2): IF NOTFOUND THEN
                   GO TO LABEL(3*(i-1)+1)
                GET <R_i>
                IF NOT(conditionals) THEN
                   GO TO LABEL(3*(i-1)+1)"
        DMLGEN(i+1)
        create DML string
           "LABEL(3*(i-1)+1): FIND DUPLICATE WITHIN
                                <R_i> USING <key>
                              GO TO LABEL(3*(i-1)+2)
            LABEL(3*(i-1)): "

     ELSE {exhaustive search needed}
        FOR (each search clause j = 1 to N) DO
```

Figure 21 - DML Generation Algorithm

```
      create DML string
         " MOVE <value> to <R_i.data item_j>"
   end FOR loop
   create DML string
      "          FIND FIRST <R_i>
                    USING <R_i.data item list>
       LABEL(3*(i-1)+2): IF NOTFOUND THEN
                    GO TO LABEL(3*(i-1))
                 GET <R_i>
                 IF NOT(conditionals) THEN
                    GO TO LABEL(3*(i-1)+1)"
   DMLGEN(i+1)
   create DML string
      "LABEL(3*(i-1)+1): FIND NEXT <R_i>
                    USING <R_i.data item list>
                    GO TO LABEL(3*(i-1)+2)
       LABEL(3*(i-1)): "

ELSE {system-supported two-variable clauses}
   <S> := CODASYL set name for relationship mapping

   IF (mapping is functional) THEN
      create DML string
         "  FIND OWNER WITHIN <S>
            IF NOTFOUND THEN GO TO LABEL(3*(i-1))
            GET <R_i>
            IF NOT(conditionals) THEN
               GO TO LABEL(3*(i-1))"
      DMLGEN(i+1)
      create DML string
          "LABEL(3*(i-1)): "

   ELSE {mapping is from member to owner}

      IF (search key is available) THEN
          FOR (each search clause j = 1 to N) DO
             create DML string
                " MOVE <value> to <R_i.data item_j>"
          end FOR loop
          create DML string
             "          FIND <R_i> WITHIN <S> CURRENT
                           USING <R_i.data item list>
              LABEL(3*(i-1)+2): IF NOTFOUND THEN
                                    GO TO LABEL(i)
                           GET <R_i>
                           IF NOT(conditionals) THEN
                              GO TO LABEL(3*(i-1)+1)"
          DMLGEN(i+1)
```

Figure 21 Continued - DML Generation Algorithm

```
            create DML string
               "LABEL(3*(i-1)+1): FIND DUPLICATE WITHIN
                         <S> USING <R_i.data item list>
                         GO TO LABEL(i+2)
                LABEL(3*(i-1)): "

        ELSE {USING clause will not work}
            create DML string
               "         FIND FIRST <R_i> WITHIN <S>
                LABEL(3*(i-1)+2): IF NOTFOUND THEN
                             GO TO LABEL(3*(i-1))
                         GET <R_i>
                         IF NOT(conditionals) THEN
                             GO TO LABEL(3*(i-1)+1)"
            DMLGEN(i+1)
            create DML string
               "LABEL(3*(i-1)+1): FIND NEXT <R_i>
                                    WITHIN <S>
                                 GO TO LABEL(3*(i-1)+2)
                LABEL(3*(i-1)): "

    end procedure DMLGEN
```

Figure 21 Continued - DML Generation Algorithm

For the i^{th} nested record type in the list, the algorithm checks to see if any conditional clause can be supported by a set access. If this fails, then it tries to find a one-variable equality clause on which to base the access. There are three cases of these. The cases and the algorithm's action for each are:

1. The key attribute of the relation appears in a one-variable clause. In this case, code is generated to access the CODASYL record by its key data item.

2. Indexed attributes appear in a one-variable equality clause. In this case, an indexed minimum cost access path is used to find the record.

3. Value attributes appear in equality clauses. Here a FIND USING statement is used to access the record. This statement automatically does the equality test on the value data item.

As a consequence of the intersection-free query, each record type is accessed from at most one previously visited record type. If the link is functional, then the FIND OWNER statement is used. If this fails, and one-variable equality clauses are present, the FIND CURRENT -- FIND DUPLICATE statements are used in order to take advantage of the USING option. Finally, if that also failed, the FIND FIRST/NEXT WITHIN SET statements are generated.

Query Restrictions for Algorithm Simplification. There are three restrictions placed on queries in order to simplify the algorithm. The first restriction is that all boolean conditions are assumed to be in conjunctive normal form. The second is that all conjunctions involve, at the most, two variables. The final restriction is that OR clauses can only specify alternate values of the same attribute.

Minor Optimizations. The given algorithm is by no means optimized, but there are a few basic improvements (11:446) that can be made that will increase the efficiency of the generated DML code. These improvements include:

1. If the conditional clauses are all true (null), then the IF (conditionals) statement does not need to be generated.
2. If a projection requests only certain data items, and not the whole record, then only the requested data items need be retrieved by the GET statement. If no items are requested (for example, passing through the record as a consequence of navigation) then the GET statement does not need to be generated at all.
3. If the functional access path is total (a mandatory/ automatic CODASYL set), then the NOTFOUND test can be

eliminated. This is because the record exists only if it has an associated link record on that access path.

4. If all members of a record type are to be accessed without restricting equality clauses, then the USING option in the FIND FIRST/NEXT statements can be eliminated.

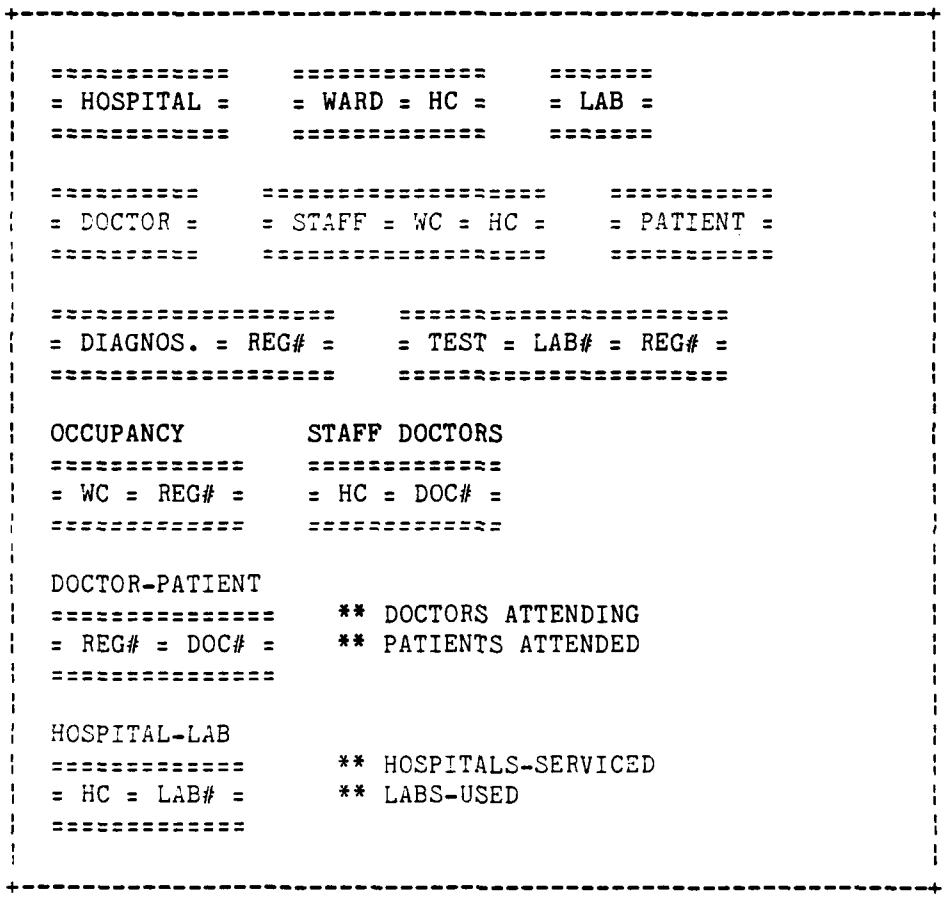


Figure 22 - Relational Schema for Medical Database (9:122)

Example Translation

In this section, an example translation is presented. The database used for this example is one that was orginally presented in Tsichritzis and Lochovsky's Data Models, but was extracted from Jones'

thesis (9:117-122). In Jones' thesis, the relational schema shown in Figure 22 was mapped from the CODASYL database shown in Figure 23. This example takes a Roth language query against the relational schema and translates it into the appropriate DML statements for the CODASYL database.

The sample query is: "Find the names of all patients who have Smith as their doctor." This query would appear in Roth form as follows:

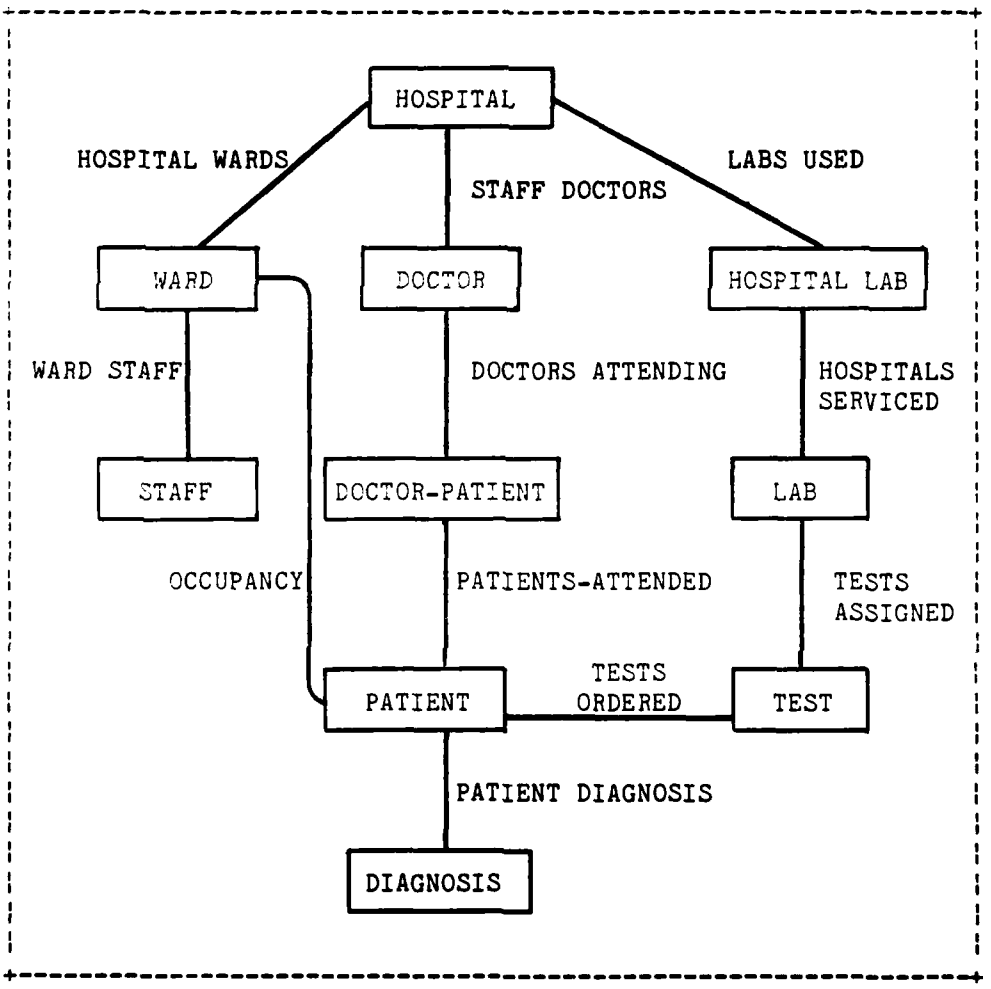


Figure 23 - CODASYL Medical Data Base (9:117)

```
SELECT ALL FROM Doctor WHERE (Doc.Name = 'Smith')
       GIVING Temp1
JOIN Temp1, Doctor-Patient WHERE (Temp1.# = Doc-Pat.#)
       GIVING Temp2
JOIN Temp2, Patient WHERE (Temp2.Reg# = Pat.Reg#)
       GIVING Temp3
PROJECT Temp3 OVER (Pat.Name)
       GIVING Temp4
```

The DML statements required to accomplish the query in the CODASYL database are:

```
MOVE 'Smith' TO Name IN Doctor
FIND ANY Doctor USING Name IN Doctor
IF NOTFOUND GO TO END
FIND FIRST Doctor-Patient WITHIN Doctors-Attending
DO WHILE (NOTFOUND = False)
       FIND OWNER WITHIN Patients Attended
       GET PatientName IN Patient
       FIND NEXT Doctor-Patient WITHIN Doctors-Attending
END WHILE
END:
```

Given the sample Roth query as input, the statements generated by the DMLGEN algorithm should be equivalent to the above DML query. The following sections briefly run through the translation process for this example, with the final generated DML statements listed in Figure 24.

Creation of Ordered List. For purposes of this example, the access characteristics of the record types in question are as follows: The **Doctor** record type has a CALC key, which is the **Doc.Name** data item, and the association between Doctors and Patients is not total. That is, a **Doctor-Attending** CODASYL set can be empty, meaning that a doctor may not currently have a patient. However, the

Patients-Attended CODASYL set will not be empty unless there are no patient records within the database.

A request to the local data directory (LNDD) returns the record type names and associated access characteristics that are needed to translate the query. The product of the Order Selection Algorithm is the following ordered list.

```
FOR EACH Doctor
  WITH access-by-identifier USING Name data-item
  WHERE KeyValue = 'Smith' DO

  FOR EACH Doctor-Attending
    WITH no-equality-clause
    WHERE Owner = Doctor AND Member = Doctor-Patient DO

    FOR EACH Patients-Attended
      WITH functional-access
      WHERE Owner = Doctor-Patient AND Member = Patient
        PRINT Patient.Name
```

Code Generation Process. The list of FOR EACH record types is then passed to the DML generation algorithm. The FOR EACH members of the list are numbered 1, 2, and 3 (recursion goes down three levels). Processing of the first level, DMLGEN(1), proceeds as follows:

1. The test for a one-variable clause is satisfied (Name = 'Smith').
2. The test for "access-by- identifier" is met.
3. The appropriate section of code is generated.
4. DMLGEN(2) is called.
5. Upon return from DMLGEN(2), generate ending code.

The second level proceeds as follows:

1. The test for a one-variable clause fails.
2. The test for functional mapping fails.
3. The test for available search-key fails, meaning code without a USING clause must be generated.
4. The appropriate section of code is generated.
5. DMLGEN(3) is called.
6. Upon return from DMLGEN(3), generate ending code.

The third level proceeds as follows:

1. The test for a one-variable clause fails.
2. The test for functional mapping is met.
3. The appropriate section of code is generated.
4. DMLGEN(4) is called, but since 4>3, this stops the recursive process. The algorithm then backs out of the levels of recursion one at a time.
5. Generate ending code.

Generated Code. Figure 24 shows the code produced by DMLGEN for this sample query. Unnecessary statements are eliminated in accordance with the optimizations mentioned above. Angled brackets (<>) denote values that are either inserted into variables or are generated by the algorithm (such as label indexes). Inspection of the generated code reveals that it is equivalent to the CODASYL query listed earlier in this chapter.

```
        MOVE <'Smith'> to <Doctor.Name> IN <Doctor>
        FIND ANY <Doctor> USING <Doctor.Name>
        IF NOTFOUND THEN GO TO <L0>

        FIND FIRST <Doctor-Patient>
             WITHIN <Doctor-Attending>
  <L5>: IF NOTFOUND THEN GO TO <L3>

        FIND OWNER WITHIN <Patients-Attended>
        IF NOTFOUND THEN GO TO <L4>
           GET <Patient.Name> IN <Patient>
  <L6>:
  <L4>: FIND NEXT <Doctor-Patient>
             WITHIN <Doctor-Attending>
        GO TO <L5>
  <L3>:
  <L0>:
        END
```

Figure 24 - Example Generated DML

Summary

This chapter detailed the process of translating a query from relational to CODASYL DML. The selection of an efficient query processing order and the generation of translated CODASYL DML were examined. Algorithms were presented for the production of an efficient ordered listing of records, and for the generation of a CODASYL query program. A sample query translation was also presented. Over the next two chapters, the design and partial implementation of similar algorithms for TOTAL, a non-CODASYL network database system, will be examined.

VIII. Relational to TOTAL DML Translation

Introduction

In the previous chapter, algorithms were presented for translating a relational query into the equivalent CODASYL DML. In this chapter, the design of similar translation algorithms is examined for a another specific network DBMS, TOTAL, marketed by Cincom Systems, Inc. The chapter begins by briefly comparing the two systems, followed by an overview of the TOTAL DML. The chapter ends with descriptions of the translation process and algorithms.

Comparison of TOTAL and CODASYL

TOTAL is a network database management system, but it differs from the CODASYL proposal in several ways. Cardenas (4:218) describes the design concept of TOTAL as being 60 to 80 percent like CODASYL, but with a DML syntax similar to IMS. The next few sections describe some of the differences between this very popular DBMS and the CODASYL model. Only a brief discussion of TOTAL DML commands is included here, with a more detailed description available in Appendix B.

Schema Terminology. First of all, the terminology of a TOTAL database schema is different from the CODASYL proposal. The differences between the two are shown in Table 3. Note that these terms may not be exactly equivalent. Usually the terms refer to the same type of entity (i.e., Item and Field being equivalent), but for others (i.e., CODASYL Set to Linkage Path) the physical implementations are different.

CODASYL Term	TOTAL Term
Data Item	Data Field
	Data Element
Record Type	Data Set
Records	Data Records
Owner Records	Master Data Sets
Member Records	Variable-Entry Data Sets
CODASYL Sets	Linkage Paths/Chains

Table 3. CODASYL and TOTAL Schema Terminology

Data Structure. Both CODASYL and TOTAL are network model DBMSs, but there are significant differences. In the CODASYL model, a record can be owner or member in any number of sets. In TOTAL, Master datasets are the owner records and Variable-Entry datasets can only be member records. In CODASYL, set members may vary in length, can be members of multiple sets, and are independent of the entry point access method. In TOTAL, Variable-Entry datasets must be of fixed length, only belong to that one set type, and can only be accessed by a chain beginning at the first or last entry points from an owner Master dataset.

Data Structure Implementation. In CODASYL, Prior and Owner pointers are optional. These pointers, plus a Next pointer, are mandatory in TOTAL. All pointer implementations in CODASYL use physical pointers, while Owner pointers in TOTAL are symbolic.

Access Methods. All records and sets in CODASYL can be accessed in three ways: Physically Serial, Random, and Direct. In TOTAL, only the Master datasets can use Random access (they can also be accessed serially), while Variable-Entry datasets must be accessed serially in a chain, either forward or backward, through a Master dataset linkage path. The only exception to this is if the actual physical location indicator for that particular data record is known.

TOTAL Data Management Language (DML). The TOTAL DML is an extension to existing programming languages, consisting of a series of CALL statements to a TOTAL interface program known as DATBAS. There are three different types of DATBAS calls, each with its own parameter list.

(1) The first, using four parameters, is for signing onto TOTAL and for opening and closing the database schema:

```
CALL DATBAS (FUNCTION, STATUS, SCHEMA, 'END.')
```

(2) The second, using seven parameters, is for accessing Master datasets:

```
CALL DATBAS (FUNCTION, STATUS, DATA-SET, CONTROL-KEY,
             ELEMENT-LIST, USER-AREA, 'END.')
```

(3) The third, using nine parameters, is for accessing Variable-Entry datasets:

```
CALL DATBAS (FUNCTION, STATUS, DATA-SET, REFERENCE,
             LINKAGE-PATH,CONTROL-KEY, ELEMENT-LIST,
             USER-AREA, 'END.')
```

The various options for the FUNCTION parameter that are used in this implementation of the query translator are listed in Table 4,

CODASYL RETRIEVAL STATEMENT	TOTAL DML FUNCTION BY DATA SET TYPE		
	MASTER	VARIABLE	BOTH
READY			SINON
FINISH			SINOF
FIND/GET	READM RDNXT	READV	

Table 4. Comparison of CODASYL AND TOTAL DML

listed under their appropriate data set type. The corresponding CODASYL statements are also listed for comparison. For a complete description of all available read-only functions and other parameters in the DATBAS call statements, see Appendix B.

TOTAL DML Generation Process

The following sections present a set of algorithms for the translation of a relational query into a program containing the requisite calls to TOTAL. The translation process breaks down into three steps: creation of the structures used by the DML generator, ordering of the structures into the optimal processing sequence, and the generation of the source code with embedded TOTAL DML.

Dataset Structure Creation

The DML generation algorithms use as input a list of data structures similar to the IQL-like list presented in Chapter VII. These structures contain the information necessary for the generation of proper DML code. This query information, obtained from the local data directory in the DDBMS, is as follows:

Query operation
Name of database--to get list of all datasets required
Name and type (master or variable) of datasets
Dataset Key
All required field names for each dataset
Size of all fields
Linkpath and reference names from the master dataset
    to variable dataset
Qualifier operators and operands (literal or fieldname)

This information is first placed into an array or list of structures that are used by the generation program. The structures are constructed as shown in Figure 25.

Ordering of the Structures

The algorithms for ordering the list are essentially unchanged from Chapter VII (and so are not presented again), with the difference

```
Dataset Name
Dataset Type (M for master, V for variable)
Dataset Key Field Name
Access Indicator (1 and 2 - READM, 3 - RDNXT, 4 - READV)
Linkpath (variable data sets only - NULL otherwise)
Reference (variable data sets only - NULL otherwise)
Number of Fields Requested (i:= 1 to N)
     Field_i
          Name
          Size
          Output Indicator (Y if field is to be printed)
Number of Boolean Qualifiers (i := 1 or 2)
     Comparison_i
          Field_i Name
          Operator
          Comparison Argument Type (literal or field)
          Qualifier Field (if field argument)
          Qualifier Literal (if literal argument)
          Compound Boolean Indicator (AND or OR)
```

Figure 25. Dataset Structure

being that there are fewer access options available in TOTAL. Master data sets are accessed in only two ways; directly (for an equality comparison on a dataset key) and sequentially (all other cases). Variable data sets must be accessed sequentially through the chain beginning with the first or last data set in the chain. (Variable data sets may also be accessed if the physical location pointer is known, but user knowledge of a pointer value would be extremely unlikely.) For simplification, variable data set access always begins with the first data set in the chain, because the average search will be the same for both forward and backward searches through the chain.

Given the limited number of access options in TOTAL, ordering the structures into the optimal access sequence is a much simpler than in CODASYL. The most efficient retrieval is for a Master dataset with an equality comparison on the dataset key. All other retrievals of Master datasets must be done sequentially. Since the retrieval of Variable datasets must be done via an access pointer from a Master dataset, the efficiency of Variable dataset retrieval is dependent upon the owning Master dataset. This access restriction of TOTAL guarantees that the critical starting dataset (see Chapter VII, Choosing the Starting Record) will be a Master dataset. If, in the remaining datasets, a dataset key is to be provided for a Master dataset by a Variable dataset field, then that Variable dataset will immediately precede the Master in the processing order. Otherwise, the deciding factor in ordering will be the presence (or lack) of comparisons on Master dataset keys.

```
procedure MAIN(ordered list of dataset structures)
  Number datasets in ordered list as 1,2,...,N
  Open GeneratedProgram File

  Create DML Code
     Label Declarations (Nx3 labels plus errorlabel)
     Type Definitions
     Declare 'N' structures for datasets retrieved
     Variable Declarations
     Declare procedures for DATBAS calls (total of N+1)
        One for SONOROFF
        One for each READV (variable dataset)
        One for each READM (master dataset)
        One for each RDNXT (master dataset)

     DATASETS := <concatenation of all database datasets>
     Open Output file
     Generate SinOnOff Call

  TOTGEN(1)

  Create DML Code
     ERRLABEL:  SinOnOff Call
                Close Output file

  Close GeneratedProgram File
  Compile and Execute GeneratedProgram File
  Display Result File
  Remove GeneratedProgram and Result Files
end MAIN
```

Figure 26. Code Generation Driver

Code Generation Algorithms

After the order selection is complete, the structures are input to the DML code generation routines. The figures on these pages present the code-generation algorithms in pseudocode. The main code-generating procedure, TOTGEN, is derived from the CODASYL algorithm in Chapter VII.

Driver Program. The main translation program, or driver, is shown in Figure 26. This may be a single program with embedded system calls or it may be a mix of system routines, such as command files, and the program with its subprocedures. A subprocedure for each generation step, such as declaration of labels and types, is assumed.

Subprocedures for DATBAS Calls. One of the more restrictive aspects of TOTAL (from a relational viewpoint) is that the parameters for each DATBAS call to TOTAL are fixed. This means that for each dataset that is to be accessed by a translated query, there must be a corresponding declared DATBAS call with the unique parameter declarations. The only way to implement this and still allow any flexibility in the queries is to declare each DATBAS call within its own subprocedure, thus "hiding" that particular DATBAS call from all of the others. There are four different types of procedures that must be generated, depending on the data set. The procedures generated would be READV, READM, and RDNXT. If a particular query involved two master datasets read by dataset key and one variable dataset, then two different READM procedures and one READV would be declared and generated.

TOTGEN DML Generation Algorithm. Once the variables and subprocedures have all been declared, the main body of the translated query program must be generated. This is handled by the TOTGEN procedure, shown in Figure 27, which calls itself

```
procedure TOTGEN(i)
  IF (i > N) THEN DO
       Output all structure contents to output file
       return {to previous level}
  endif

  IF (Master Dataset) THEN
     IF (one-variable equality clause) AND
        (Clause is a Dataset Control Key) THEN

           create DML string
             "READM_i(<SearchKey_i>,<UserArea_i>)

               IF NotFound THEN GO TO LABEL(3*(i-1))
               IF (STATUS <> '****') THEN
                   GO TO ERRLABEL
               <Structure_i> := <UserArea_i>"

         IF (Structure_i.value(j) == qualifier)
                THEN (successful)
         IF (not successful) THEN GO TO LABEL(3*(i-1))"

        TOTGEN(i+1)

           create DML string
             "LABEL(3*(i-1)): "

     ELSE {sequential search - not dataset key}

           create DML string
              "Qualifier_i := 'BEGN'
               RDNXT_i(<Qualifier_i>,<UserArea_i>)
LABEL(3*(i-1)+2): IF QUALIFIER == 'end of chain' THEN
                    GO TO LABEL(3*(i-1))
               IF (STATUS <> '****') THEN
                   GO TO ERRLABEL
               <Structure_i> := <UserArea_i>"

         IF (Structure_i.value(j) == qualifier)
                THEN (successful)
         IF (not successful) THEN GO TO LABEL(3*(i-1)+1)"

           TOTGEN(i+1)

           create DML string
"LABEL(3*(i-1)+1): RDNXT_i(<Qualifier_i>,<UserArea_i>)

               GO TO LABEL(3*(i-1)+2)
```

Figure 27. TOTAL DML Generation Algorithm

```
LABEL(3*(i-1)): "

   ELSE {Variable datasets - read chain in sequence}

            create DML string
               "<SearchKey_i> := <Dataset(i-1).KeyValue>
                READV_i(Reference_i,<SearchKey_i>,
                        <UserArea_i>)

LABEL(3*(i-1)+2): IF Reference_i == 'END.' THEN
                     GO TO LABEL(3*(i-1))
                  IF (STATUS <> '****') THEN
                     GO TO ERRLABEL
                  <Structure_i> := <UserArea_i>"

          IF (Structure_i.value(j) == qualifier)
                THEN (successful)
          IF (not successful) THEN GO TO LABEL(3*(i-1)+1)"

            TOTGEN(i+1)

            create DML string
"LABEL(3*(i-1)+1): READV_i(Reference_i,
                           <SearchKey_i>, <UserArea_i>)

                  GO TO LABEL(3*(i-1)+2)
LABEL(3*(i-1)): "

   end procedure TOTGEN
```

Figure 27 Continued. TOTAL DML Generation Algorithm

recursively in order to build the proper processing order into the program. This is essentially the same as the DMLGEN algorithm presented in Chapter VII. In the figure, the subscript "i" signifies the declaration of variables for the i^{th} level of recursion, being the same as the number of the data set that is being retrieved. The angled brackets (<>) signify values (variables) that are generated for that particular call of TOTGEN.

Summary

In this chapter, the design of a translator program for TOTAL, a non-CODASYL network data base management system, was discussed. The significant differences between TOTAL and the CODASYL proposal were pointed out, and algorithms (modified from the previous chapter) for the generation of TOTAL DML were presented. In the next chapter, the partial implementation and testing of a relational to TOTAL DML translator program is presented.

IX. A Partial Implementation of the TOTAL Translator

Introduction

This chapter describes the partial implementation of the relational to TOTAL DML translation algorithms of the previous chapter, and the testing of sample queries using the AFIT Data Base (AFITDB).

The chapter is organized into three main parts. The first part details the implementation of the translation software. The second discusses the AFITDB, and details the portion of the database used to create a set of sample queries. The last part of the chapter presents the sample queries used to test the translator program and analyzes the test results.

Translation Software Host Machine and Language

Implementation of the translation software was done on the VAX-11 that also hosted the AFIT TOTAL DBMS. The translator algorithms presented in the previous chapter were implemented using the C programming language. The DML generator portion of the program generates a Pascal language program with embedded TOTAL DML statements which, after being compiled and linked to TOTAL, actually execute the query. The choices of the VAX and C/Pascal as the host machine and languages were based on the following criteria and constraints:

1. The DDBMS network link to the VAX had not been successfully implemented, making distributed access impossible.
2. The LSI-11 microcomputers that compose the DDBMS nodes were restricted in their memory capacity. It was doubtful that the LSI's could accommodate network protocol software, LNDD and ECNDD software, and the translator software all at once.

3. **C** was the most efficient language available on the VAX (the others being Pascal and Fortran) for the handling of character strings, which is the major task of the DML program generation portion of the translation process.

4. The method of interfacing Pascal to the TOTAL DATBAS program (written in FORTRAN) was already known. Development of an interface to **C** would be a possibly lengthy proposition.

Translator Limitations and Assumptions

The translation software was implemented using several limiting design decisions and assumptions. The major assumptions and limitations are pointed out in the following subsections, along with the criteria on which these decisions were based.

Query Input. The information used by the translation program is assumed to be the information passed to the local DBMS by the LNDD at the host node in the network. The actual parsing of the query would already be done at this point, and only the information returned from the LNDD would be relevant to the translator. This decision was based on the fact that the LNDD and ECNDD, which are critical to parsing, are located on the DDBMS nodes (LSI-11 microcomputers), not the local DBMS host machines. As such, query parsing would most likely be done on the LSI-11s, not the VAX. The query information is assumed to be sent as a file (known as QUERY.DAT) to the VAX, which would then be read by the translator.

Multiple Databases. One early decision was that the query translator program must be capable of handling multiple databases resident on the TOTAL DBMS. Multiple databases are a common DBMS

occurence. It is more unreasonable to expect a DBMS to contain only one database. The method of handling this was to assume that the first information present in the query file would be the six-character database name. The translator program then uses this information to open an input file containing the database schema. This schema information is required by TOTAL when signing on to the database (see Appendix B). For example, the schema for the AFITDB was located in file AFITDBSC.DAT (SC standing for schema).

Dataset Processing Order. The most significant omission from the algorithms presented in the previous chapter was the one for the ordering of datasets into the most efficient processing sequence. It was assumed, for the moment, that queries would be input in the most efficient processing sequence. The immediate concern was to implement and test software that actually could translate queries. The efficiency of the queries can be addressed later, without affecting the previously implemented code generation software.

Queries Allowed. All queries to TOTAL are assumed to be path, not tree, queries. This is a corollary to the previous decision not to implement the query processing order algorithm. When that algorithm is implemented, the possibility of allowing tree queries (and the associated implementation of more complicated ordering algorithms) can be addressed.

Boolean Qualifiers. The implementation of multiple boolean qualifiers on the datasets in a query became an issue almost as

involved as the translation itself. In order to simplify the process of generating qualifier code, the number of qualifiers on a particular dataset was limited to two. For Read Unique Master (READM) datasets, one extra qualifier was allowed in addition to the equality comparison on the database key (see the previous chapter). For all other retrievals, two qualifiers were allowed, with the **AND** or **OR** logical connectives, as shown below:

<qualifying condition 1> AND/OR <qualifying condition 2>

This allows range specifications (i.e., retrieve all workers older than 20 AND less than 35) and multiple selection criteria (i.e., all parts supplied by Jones OR supplied by Smith) on individual datasets retrieved by READV or RDNXT calls to TOTAL. Although selection criteria on a single dataset is limited to two qualifiers, a query involving more than one dataset still can have several qualifying conditions. A join involving three datasets could possibly have six qualifiers, two for each dataset.

Another capability was to allow the selection of data records based on a subset of the dataset key. For example, in the AFITDB, one course's dataset key is EENG650. A query involving the selection of all electrical engineering courses would use only the 'EENG' portion of the key. In this case, the RDNXT statement would be used, rather than the READM.

Maximum Datasets in a Query. The list of structures presented in the previous chapter was implemented as a fixed-length array, thus

placing an arbitrary limit on the number of datasets that could be involved in the query. The limit chosen was seven datasets, or the "Hrair Limit". The Hrair limit is the maximum number of entities that the human mind can easily comprehend, and is generally considered to be seven items, plus or minus two. Other limits placed on the structures are a maximum of 40 fields in a dataset, two qualifier comparisons (explained above), and only one linkpath and reference allowed per variable dataset. The latter could (and should) be expanded when the processing order algorithm is implemented, since multiple access paths would then be possible.

Translator Input and Output

All software development was done in a single directory on the VAX. In this directory, there are two input files to the translation program, AFITDBSC.DAT and QUERY.DAT. There are four output files, TCODE.PAS (the generated program), TCODE.OBJ and TCODE.EXE (compiled and linked versions of TCODE), and QRESULT.DAT, the query result file created by running TCODE. The following sections examine the two input files and the output result file.

AFITDBSC.DAT File. This file contains the schema information for the AFIT database. As more TOTAL databases are tested, additional <database name>SC.DAT files will have to be created. The information present in the file is as follows:

Size of Schema Declaration
Number of Following Lines in Schema Declaration
N Schema Declaration Lines

The first two items of information are used simply to ease the implementation of the translator software. The schema size is used early in the translator to define the buffer size for the schema used in the SINON/SINOF to TOTAL. The number of lines counter is to allow the proper formatting of the schema declaration in the generated SONOROFF routine. The declaration lines contain the required information (program name, database name, access mode and field names) as noted in Appendix B. For purposes of illustration, the first schema line of the AFITDB is as follows:

'GENTCODEAFITDBRDONLYNLFACTSHREXXXXDEPTSHPEXXXX'

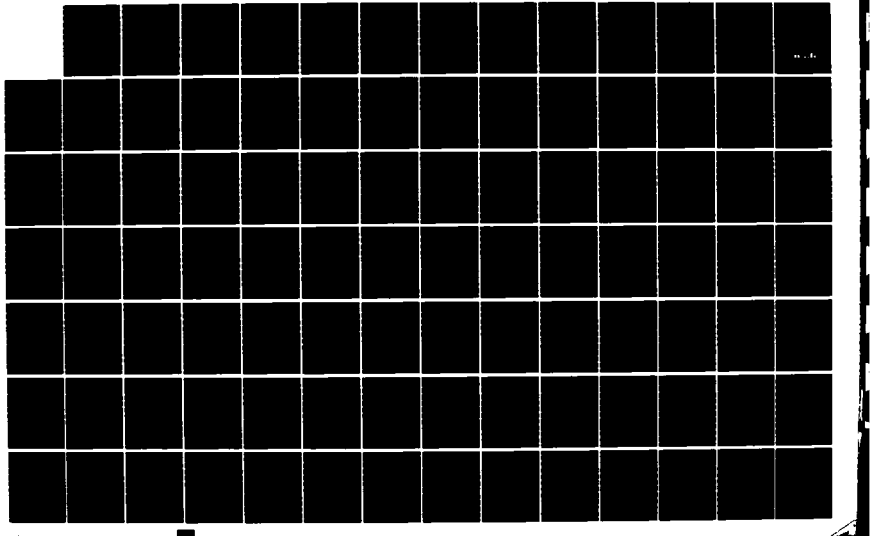
QUERY.DAT File. This file contains the query information that is returned by the LNDD in the DDBMS. The format of the information here is similar to the format being used by Wedertz in his thesis (20). The exact format of QUERY.DAT is:

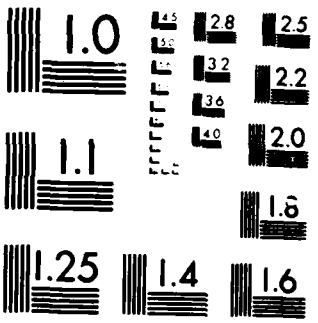
Database Name (AFITDB)
Number of Datasets (N) in Query
1st Dataset Information
-
-
-
Nth Dataset Information

There are two different formats for the dataset information, depending on the dataset type. These are shown on the next page.

AD-A164 013 THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K H MAHONEY 2/3

UNCLASSIFIED DEC 85 AFIT/GCS/ENG/85D-7 F/G 9/2 NL





MASTER DATASETS	VARIABLE DATASETS
Dataset Name (4 character)	Dataset Name
Type Indicator (M)	Type Indicator (V)
Control (Key) Field Name	Control Field Name
Number of Fields Requested	Linkpath Name (8 character)
i^{th} Field	Reference (4 character)
Field Name (8 character)	Number of Fields Requested
Field Size (in characters)	i^{th} Field
Output Indicator (Y or N)	Field Name
Number of Qualifiers (0,1,2)	Field Size
i^{th} Qualifier	Output Indicator
Field Name (being compared)	Number of Qualifiers
Comparison Operator	i^{th} Qualifier
Argument Type (F or L)	Field Name
Conjunction	Comparison Operator
	Argument Type
	Conjunction

Explanation of some of the values may be necessary. The output indicator determines if the retieved field is to be output to the result file. The fields actually requested by the user's query are denoted by a 'Y' for output, and the other fields needed (as determined by the LNDD) are marked with a N, so they are not seen in the output result file.

The qualifier field name is the field in the dataset that is being compared against. The Argument Type is 'L' for a user-supplied comparison literal, and 'F' denotes a comparison against a field from another dataset (as in a natural join). The comparison operator must be one congruent to those used by Pascal. The operators that must be used are:

'=' - Equal To	'<>' - Not Equal To
'<' - Less Than	'<=' - Less Than Or Equal To
'>' - Greater Than	'>=' - Greater Than Or Equal To

The conjunction/disjunction field will be 'XXX' if the number of qualifying comparisons is 0 or 1. If it is 2, then the first qualifier will have a 'AND' or 'OR' in the field, and the second qualifier will have a 'XXX' (since there is no third field).

QRESULT.DAT File. The query result file is written by the generated program (TCODE) when it is executed. The results are written in a simple format that separates each "tuple". The requested output from each of the datasets involved in the query is output on a separate line, with another line separating each distinct data aggregate (tuple) in the result. For example, if a query requests a student name from one dataset, a course title from another, and the grade from a third, a resulting tuple might be:

```
Smith, John A.
Advanced Database Systems
A-
```

This query result file would then be transmitted back to the requesting DDBMS node for further processing, possibly to be joined or unioned with the results from a partitioned query to another database.

Processing Sequence

The processing sequence of the translator program, TRANS.C, basically consists of a series of passes down the array of dataset structures. First, the input from QUERY.DAT is read in, building the array of datset structures for use by the remainder of the program. The access characteristics of each dataset are then analyzed, with an

access type classification of 1, 2 (both READM access), 3 (RDNXT), or 4 (READV) being assigned to each dataset.

The code generation process now begins. After opening statements have been generated, the second pass down the dataset array checks the size of each dataset field and computes literal sizes in order to create the list of sizes for buffer-type declarations in the Pascal program. Another pass down the array creates record-types for query output. A fourth pass through the datasets and all fields generates the variable declarations for the program. The fifth pass down the dataset array generates one subprocedure (for the unique DATBAS call) for each dataset in the query. It is at this point (in the SONOROFF procedure) that the database schema file (i.e., AFITDBSC.DAT) is read. The sixth pass is made by the recursive module that generates the body of the Pascal program, with other, partial, searches of the array occurring as needed when computing the fields required in comparison qualifiers. The seventh complete pass is made when the recursion stops and output statements are generated.

At this point, the final code statements are generated and the output file, TCODE.PAS is closed. What remains now is to compile and link the program to TOTAL, and then execute it. Running the TCODE executable code is what actually creates the result file.
For a detailed description of the program, the reader is referred to Appendices E, F, and G, which contain, respectively, the Data Dictionary, Structure Charts, and source code listings.

The AFIT Data Base (AFITDB)

The TOTAL database chosen for program testing was the AFIT Data Base (AFITDB), which was designed to handle the scheduling of classes, maintain student, faculty, and thesis information, and to track order information on textbooks. The AFITDB is composed of 37 separate datasets (15 Master files and 22 Variable files) and is resident on the AFIT Information Sciences Laboratory's VAX 11/780 minicomputer.

The main difficulty encountered with the AFITDB was the fact that the database is still basically in a state of infancy. Most of the 37 datasets presently have very little, if any, information in them, and several of the usable datasets do not have linkpaths connecting them to other datasets. The usable portion of the database that was utilized for query translation testing is outlined below.

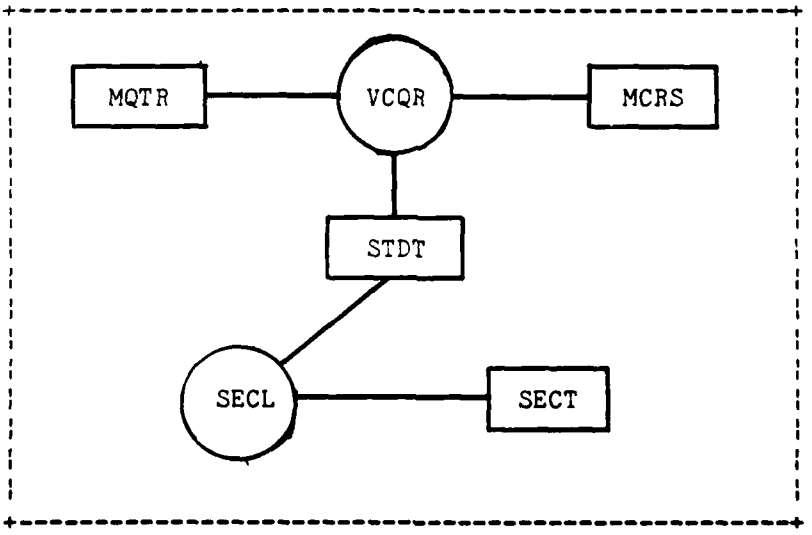


Figure 28. TOTAL Schema for Test Database (Subschema of the AFITDB)

Test Subschema of the AFITDB. The portion of the AFITDB that was used in this query consisted of four Master datasets and two Variable datasets, for a total of six datasets in the subschema. A diagram of the subschema showing the TOTAL relationships of the datasets is shown in Figure 28.

There were other Master datasets containing information that were not included in the test subschema. This was because the Variable dataset linkpaths for the Masters were not implemented, thus leaving them isolated. A simple select, project, or non-supported join could be run on these Master datasets, but those queries could also be tested on the selected datasets. As such, the isolated datasets were removed from consideration.

Description of Subschema Datasets. The six datasets selected for the test subschema are listed below, with their respective fields that actually contained information. The four-letter code names shown are the actual TOTAL names for the datasets and fields. Several of the datasets actually possess more fields than are shown here, but in these cases, invalid or no information was present in the omitted fields. A specific case involved the the STDT (Student) Master dataset. The STDT dataset actually has 32 separate fields, encompassing a wide range of information such as home address, spouse names, and military date of rank. However, only the Student Name, Rank, and Control fields contained any information. As such, only these fields are shown below.

STDT - Student Master Dataset
CTRL - dataset control field (student SSAN)
NAME - student name
RANK - military/civilian rank
LKSE - linkpath to the SECL variable dataset
LKCQ - linkpath to the VCQR variable dataset

SECT - Student Section Master Dataset
CTRL - dataset control field (section number)
NRSN - number of students in the section
LKSE - linkpath to the SECL variable dataset

MCRS - Courses Master Dataset
CTRL - dataset control field (course number)
CRHR - credit hours
LCHR - lecture hours
LBHR - lab hours
SZLM - class size limit
TITL - course title
LKCQ - linkpath to the VCQR variable dataset

MQTR - Quarters Master Dataset
CTRL - dataset control field (quarter/year)
STDT - start date
SPDT - stop date
LKCQ - linkpath to the VCQR variable dataset

VCQR - Variable Course-Quarter Dataset
CODE - mode control (required for retrieval/update)
NMBR - course number
IDEN - quarter-year
SSSN - SSAN of student enrolled in class

SECL - Section Leader Variable Dataset
SECT - section number
STDT - SSAN of student in section

The conversion of the TOTAL datasets for this subschema is not very difficult. The schema conversion algorithms proposed by Jones (9:117) do not apply exactly to the TOTAL data model (due to the absence of retention and membership classes in TOTAL) but do provide a workable solution. The two Variable datasets become "member of" type relations that are composed completely of foreign keys from the Master datasets (relations). The resulting relational schema is shown in

Figure 29. This is the schema that is used for the test queries presented in the next section.

Test Query Translations

Once the translation software was implemented on the VAX, a set of sample queries was run to test the operation and efficiency of the both the C translator/generation program and the generated Pascal program. Several queries resulted in the discovery of errors in the generation program, mostly due to unexpected query

STUDENT RELATION

| Student SSAN | Name | Rank |

SECTION RELATION

| Section Number | Number of Students |

COURSE RELATION

| Course Number | Title | Class Size | Credit Hours |

QUARTER RELATION

| Quarter-Year | Start Date | Stop Date |

ENROLLED-IN RELATION

| Course Number | Quarter-Year | Student SSAN |

MEMBER-OF-SECTION RELATION

| Section Number | Student SSAN |

Figure 29. Relational Schema for Test Database

combinations, but these were easily corrected. Other problems that were uncovered dealt with the problem of generating correct Pascal code, given that that language is very unforgiving in terms of type declarations. This is the reason that the first part of the translator program does nothing but generate buffer type declarations to use throughout the remainder of the Pascal program.

The final set of queries shown here were selected because they represent a wide, but normal, range of queries that could be expected in the system. There are seven sample queries, involving from one to five of the six datasets in the schema. The query is first presented in the Roth relational query format (even though the Roth query was not actually used in translation), and then the TOTAL processing sequence that is assumed by the QUERY.DAT file is presented. The actual QUERY.DAT, TCODE.PAS, and QRECULT.DAT files for each test are included in Appendix H.

First Query. This query requests the courses that student "Mahoney" took in the Winter 1985 quarter, and is enrolled in for the Fall 1985 quarter. The Roth Query is:

```
SELECT ALL FROM Student WHERE (Name = 'Mahoney')
     GIVING Temp1
JOIN Temp1, Enrolled-In WHERE (Temp1.SSAN = Enrolled-In.SSAN)
     GIVING Temp2
SELECT ALL FROM Temp2 WHERE (Temp2.Quarter-Year = 'FA85') OR
     (Temp2.Quarter-Year = 'WI85') GIVING Temp3
JOIN Temp3, Course WHERE (Temp3.Number = Course.Number)
     GIVING Temp4
PROJECT Temp4 OVER (Student.SSAN, Student.Name, Course.Number,
     Course.Title) GIVING Temp5
```

The processing sequence assumed by QUERY.DAT is:

```
STDT - RDNXT using Name = 'Mahoney' as qualifier
VCQR - READV using linkpath LKCQ from STDT
MCRS - READM using course number from VCQR as key
```

Second Query. This query asks for the name and rank of all AFIT students with last names that begin with D, E, or F. The Roth query is:

```
SELECT ALL FROM Student WHERE (Student.Name > 'C') AND
     (Student.Name < 'G') GIVING Temp1
PROJECT Temp1 OVER (Student.Rank, Student.Name)
     GIVING Temp2
```

The processing sequence assumed by QUERY.DAT is:

```
STDT - RDNXT with qualifiers (Name > 'C') AND (Name < 'G')
```

Third Query. This query requests the name and rank for the student with '062084021' as his/her SSAN. The Roth query is:

```
SELECT ALL FROM Student WHERE (Student.SSAN = '062084021')
     GIVING Temp1
PROJECT Temp1 OVER (Student.Rank, Student.Name)
     GIVING Temp2
```

The processing sequence assumed by QUERY.DAT is:

```
STDT - READM with dataset key qualifier '062084021'
```

Fourth Query. The request here is for all the courses that the Electrical Engineering Department offers. The Roth query is:

```
SELECT ALL FROM Course WHERE (Course.Number = 'EENG')
     GIVING Temp1
PROJECT Temp1 OVER (Course.Number, Course.Title)
     GIVING Temp2
```

The processing sequence assumed by QUERY.DAT is:

```
MCRS - RDNXT because qualifier is a subset of the database key
```

Fifth Query. This query requests the names of all the students that are in section GCS-85D. The Roth query is:

```
SELECT ALL FROM Member-of-Section WHERE
     (Section Number = 'GCS-85D') GIVING Temp1
JOIN Temp1, Student WHERE (Temp1.SSAN = Student.SSAN)
     GIVING Temp2
PROJECT Temp2 OVER (Student.Name)
     GIVING Temp3
```

The processing sequence assumed by QUERY.DAT is:

SECT - READM where key is 'GCS-85D' (This actually came out as RDNXT, because the literal is only seven characters and the field is eight, so a subset of the key was assumed by the program)
SECL - READV using linkpath LKSE from SECT
STDT - READM using SSAN from SECL as key

Sixth Query. This query retrieves all the GCS-85D students that are enrolled in the Fall 1985 offering of MATH555, plus the quarter-year. The Roth query is:

```
SELECT ALL FROM Enrolled-In WHERE (Course Number = 'MATH555')
     AND (Quarter-Year = 'FA85') GIVING Temp1
JOIN Temp1, Student WHERE (Student.SSAN = Temp1.SSAN)
     GIVING Temp2
SELECT ALL FROM Member-of-Section WHERE (Number = 'GCS-85D)
     GIVING Temp3
JOIN Temp2, Temp3 WHERE (Student.SSAN = Member-of-
          Section.SSAN) GIVING Temp4
PROJECT Temp4 OVER (Student.Name, Enrolled-In Quarter-Year)
     GIVING Temp5
```

The processing sequence assumed by QUERY.DAT is:

SECT - READM using 'GCS-85D' as key (Actually became RDNXT for the same reason as outlined above)
SECL - READV using linkpath LKSE from SECT
STDT - READM using SSAN from SECL as key
VCQR - READV using linkpath LKCQ from STDT with additional qualifiers NMBR = 'MATH555' and IDEN = 'FA85'

Seventh Query. The last query requests the names of all GCS-85D students with last names beginning with 'A' through 'J' that are taking a MATH deparment course in Fall 1985, with the titles for the courses that they are enrolled in. This is the query that uses the maximum (five) number of datasets of all the test queries. The Roth query is:

```
SELECT ALL FROM Enrolled-In WHERE (Course Number = 'MATH') AND
     (Quarter-Year = 'FA85') GIVING Temp1
JOIN Temp1, Student WHERE (Student.SSAN = Temp1.SSAN) AND
     (Student.Name < 'L') GIVING Temp2
SELECT ALL FROM Member-of-Section WHERE (Number = 'GCS-85D)
     GIVING Temp3
JOIN Temp2, Temp3 WHERE (Student.SSAN = Member-of-
          Section.SSAN) GIVING Temp4
JOIN Temp4, Course WHERE (Temp4.Course Number = Course.Number)
     GIVING Temp5
PROJECT Temp5 OVER (Student.Name, Enrolled-In Quarter-Year,
     Course.Title) GIVING Temp6
```

The processing sequence assumed by QUERY.DAT is:

```
SECT - READM using 'GCS-85D' as key (Actually became RDNXT
               for the same reason as outlined above)
SECL - READV using linkpath LKSE from SECT
STDT - READM using SSAN from SECL as key with additional
               qualifier NAME < 'L'
VCQR - READV using linkpath LKCQ from STDT with additional
               qualifiers NMBR = 'MATH' and IDEN = 'FA85'
MCRS - READM using NMBR from VCQR as key
```

Analysis of Query Translation and Execution

All of the above queries were run over a two day period when usage of the VAX was at a low level. Stopwatch timing was done on each of the steps of query translation and execution: translation of the query file, compilation of the generated code, linking the code to the TOTAL DBMS, and executing the query. The reasoning was to see

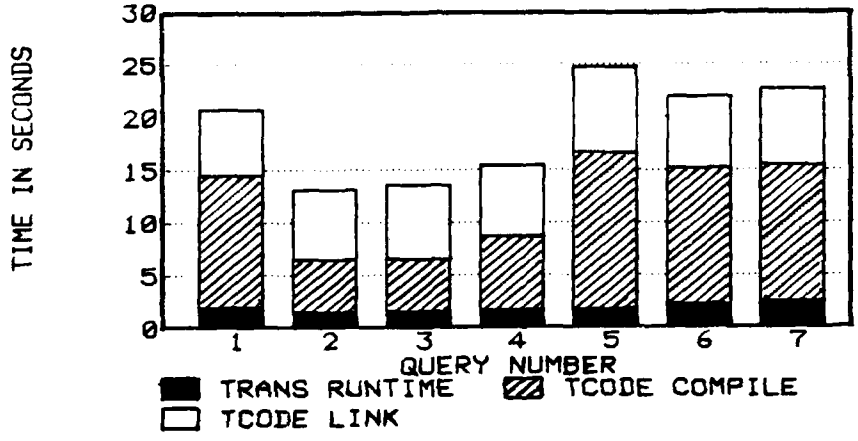


Figure 30. Code Generation and Compilation Times

approximately how long it would take to actually process a query in a heterogeneous DDBMS.

A graphical illustration of the results is included to aid in analyzing the query execution. The graph of the first three query translation steps is shown in Figure 30, the execution time of the fourth step of each query is illustrated in Figure 31, and the total

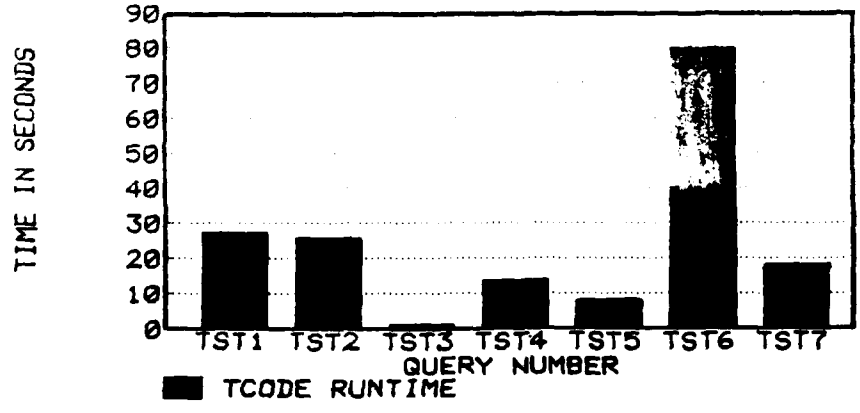


Figure 31. Translated Code Execution Times

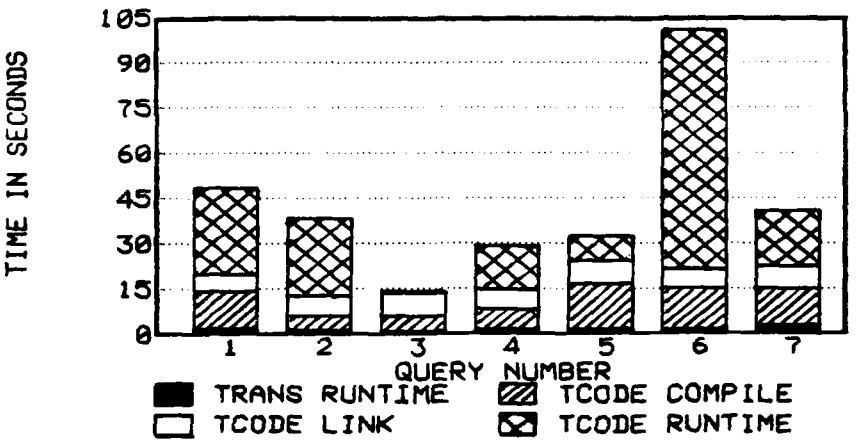


Figure 32. Total Processing Time For Query Execution

processing time is shown by the graph of Figure 32. The actual times for each processing step are given in Table 5.

Analysis. The time it takes for the query translation process obviously differs by a wide margin from query to query. The average total execution time from start to finish is 43.8 seconds, but the

QUERY NUMBER	TRANS RUNTIME	TCODE COMPILE	TCODE LINK	TCODE RUNTIME	TOTAL QUERY PROCESSING TIME
1	2.12	12.52	6.1	27.68	48.42
2	1.51	5.13	6.51	27.68	38.9
3	1.46	5.14	7.09	1.12	14.81
4	1.82	7.09	6.61	13.67	29.19
5	1.8	15.01	8.09	7.9	32.8
6	2.23	13.21	6.48	79.53	101.45
7	2.59	12.89	7.23	18.44	41.15

Table 5. Query Processing Time (in seconds)

sample standard deviation is 25.5 seconds. There is no way that one can draw any conclusions about the efficiency of such a query translator in a heterogeneous DDBMS from such a small sample, but there are some points that can be discerned from just seven queries. First of all, the length of the first three steps will be fairly equal, even for the most involved queries. The generation and linking steps are almost identical from query to query. The main variation in time is for the second step (compilation) where the average is 10.14 seconds, with a standard deviation of 3.88. Most queries will average around 20 seconds to finish the creation of the executable code. What does vary widely is the time it takes to actually execute the code.

Most of the execution times did not reveal any surprises. Query 3, where a single record was retrieved by use of the dataset key, was extremely fast, running in just over a second. The speed of this fastest query was expected. However, the slowest query, Number 6, was not anticipated. It ran for over 79 seconds, nearly three times as long as the next longest query. Complexity of the query would be the obvious answer, but this query did not involve the highest number of datasets. That was Query 7, which used five datasets, and which ran only 18.44 seconds. Why this discrepancy? It appears that the difference is in where the qualifications appear in the processing order of the query. In Query 6, the qualifications appeared in the last few datasets of the query. In Query 7, the addition of a qualification on the second dataset (STDT) sharply reduced the required amount of sequential searching.

Conclusions. Few conclusions can be drawn from such a small sample of queries, but one significant one can be. It is apparent that the criteria for the ordering of datasets in a query (which was omitted from this partial implementation) should be expanded to include an analysis of where the qualifications lie in the processing order. Knowing the relative size of the datasets would also be useful. The combination of these two factors would help to reduce the execution time of the translated program, which is obviously the key to the length of time it takes to process a query. This is a trade-off, since a complex process-ordering algorithm will add time to the initial translation, but the time saved in the execution of the query will probably exceed any additional cost from the algorithm.

Summary

In this chapter, the partial implementation of a query translator for the TOTAL DBMS was presented. The choice of the implementation machine and language was explained, and the translation algorithm, input, output, and assumptions and limitations of this particular implementation were discussed. The test data base was then presented, and seven sample queries were defined in the Roth relational format. These sample queries were translated and executed and the results of a stopwatch timing of the process were presented. The chapter then finished with an analysis of the timing results and drew one major design conclusion about query processing. The next chapter concludes this thesis, and offers some direction for future research in the area of heterogeneous distributed databases.

X. Results and Conclusions

Introduction

In the nine preceding chapters of this thesis, the many issues and considerations of implementing a global query language for a heterogeneous distributed database system were presented. Several methods of dealing with the problems and tasks of such a language were presented, but there were probably far more questions raised than answered. This final chapter will first try to tie the f ngs and accomplishments of this thesis together. It will then advocate certain areas that are considered worthy of further research, and will finish with a last few observations and conclusions.

Overview of the Thesis

This thesis could be considered as being divided into three distinct sections. The first section, Chapters 2, 3, and 4, addressed the problem of global query management. A specific model, the Database Prism, was advocated as the best approach. This was followed by an extensive discussion of the types of data partitioning that would likely be present in a distributed database built over existing local databases. This partitioning issue is particularly relevant, considering the task of decomposing global relational queries into local database queries. The idea was advocated that not only the global data model should be considered as being relational, but that the local logical databases should be viewed by the global system as being

relational. This would aid in the decomposition of queries, allowing them all to be based on the relational model.

The second part of the thesis, Chapters 5, 6, and 7, dealt with the task of translating these relational queries against the local logical model into the data manipulation language of the underlying database system. Constraints on the relational, hierarchical, and network model that were originally proposed by Jones (9) were expanded and formalized. Algorithms for the translation of relational queries into hierarchical and network DML were presented, and a sample translation was shown.

The final part of the thesis, Chapters 8 and 9, dealt with the specific implementation of such a query translation program. A translation algorithm for translating relational queries into the DML of the TOTAL database management system was designed, and a partial implementation of the design was written and tested. Results of the translator were analyzed, and some additional conclusions were reached.

Accomplishments

This thesis contained only a cursory view of the global query management problem, but it did present a detailed analysis of the very real problem of data partitioning and redundancy. This problem will almost always need to be addressed when creating a global distributed database over existing local databases. Ten different classes of partitioning were presented, and the ability to deal with these classes will

most likely need to be incorporated in any future development of a global query manager for the DDBMS.

The data partitioning problem is significant, but the real accomplishments of this thesis were in the more localized problem of query translation. Specific algorithms were researched and advocated for the translation of relational queries into IMS and CODASYL DML. The CODASYL algorithm was then modified to enable the translation of relational queries into the IMS [illegible] DAPLEX DBML.

The relational to DAPLEX translation software was partially implemented on a VAX-11, using C as the programming language. This translator is presently isolated from the remainder of the DDBMS, which is located on a set of LSI-11 microcomputers. However, the translator program has been designed to deal with Local Data Directory information, and as such, it should be fairly easy to interface with the DDBMS once the communications link between the VAX and the LSI-11s has been implemented.

The accomplishments of this thesis can be summarized into six main areas as follows:

1. A model (Database Prism) for dealing with the global query management problem was advocated and discussed.

2. Ten specific classes of partitioned and redundant information were defined, and processing sequences for proper data recomposition were outlined.

3. The schema constraints orginally proposed by Jones were further defined and formalized.

4. Algorithms for the translation of local relational queries into hierarchical and network DML were presented.

5. A new algorithm for the translation of relational queries into TOTAL DML was developed and presented.

6. A partial implementation of the TOTAL translator was developed and tested on a VAX-11 minicomputer.

Recommendations For Further Research

Although there is a great deal of information contained in this thesis, it still has barely scratched the surface of the heterogeneous distributed data base problem. This problem is one that will require the research and efforts of the academic community, as most indications are that the commercial computer industry is ignoring the problem in favor of designing new, homogeneous, distributed systems. Specific areas of continued research and follow-on efforts to this thesis should include:

1. Develop the global query manager. This thesis has implemented the portion of the DDBMS that deals only with the translation of local relational queries. Capt Wedertz's thesis dealt only with the Data Directories. There is a pressing need to develop the system for the decomposition of global queries, routing of the resulting local queries, and the recomposition of the distributed results. This obviously will require a major effort, most likely requiring several different theses.

2. Develop the use of SQL as the global query language instead of the present Roth language. SQL is rapidly gaining support within the Dept. of Defense to become the single approved query language for DOD database systems. Since the query translator is designed for Data Directory information, effort for conversion to SQL should be minimal.

3. Finish the implementation of the TOTAL translation software. Specifically, implement the portion of

the software that orders the dataset structures into the optimal processing sequence. The ability to deal with multiple boolean conditions is also an area that should be expanded.

4. Investigate the possibility of developing translators for other DBMSs. TOTAL is the only non-relational system at AFIT, but research into an IMS-type hierarchical language and the CODASYL model are the next logical steps in query translation.

5. Develop a relational front end to the query translator on the VAX. This would allow the input of relational queries from terminals on the VAX itself, rather than input from the LSI-11s. This would require development of a local query parser and data directory.

Final Conclusions and Observations

Query translation in a heterogeneous distributed database system is a very real problem for business, for government, and particularly for the Department of Defense. There is a pressing need to deal with the requirement of accessing several different databases, and designing global schemas over the present information is one approach to dealing with this need. However, the work on this thesis has indicated that this solution is not without its limitations. For example, the average time for translating and executing a query on the system was over 40 seconds, approximately half being the translation overhead and half being the actual query execution time. Admittedly, the software implemented is by no means optimal. However, this observation must be balanced by the realization that the tasks of parsing of the initial query, accessing the data directories, transmitting the local queries and results, and recomposing the

local results into a global format are not part of those 40 seconds. Still, half of the time was spent in data retrieval by TOTAL, which is going to occur no matter if the query starts out in TOTAL DML or not.

However, even if the translation of queries is not currently a particularly responsive solution, it is the best approach that is presently available, short of the actual conversion of the underlying local databases into a homogeneous system. It appears to be the only way that the "ad hoc" qualities of relational query languages can be preserved. The last few months of effort that were put into this thesis have shown that the translation of global queries into a different underlying DBMS query language is indeed possible. However, this effort has just scratched the surface. Future AFIT research into distributed databases, both heterogeneous and homogeneous, will hopefully continue to expand the body of knowledge concerning database systems. The task for the immediate future then becomes to take the present idea of a DDBMS a step further and produce a truly responsive system that can handle the ever-growing supply of information that is needed by all phases of modern society.

Appendix A: Glossary of Terms

Access Path - The sequence of physical connections present from one set of records to another in network databases. They are used to provide the means of traversing through the database.

AFIT - Air Force Institute of Technology

AFITDB - The AFIT Data Base, a database containing information on the students, faculty, courses, and facilities at AFIT.

CNDD - Centralized Network Data Directory, that contains the locations of all data items in the DDBMS.

CODASYL - Conference on Data Systems Languages, which produced a "standard" for network data bases.

DBMS - Data Base Management System, a software module executing at host computers that organizes and retrieves data.

DDBMS - Distributed Data Base Management System, software modules executing with network protocol modules to combine databases together over network lines into a larger, single database.

DBTG - Data Base Task Group, a CODASYL working group that developed the "standard" network database proposal known as the DBTG database.

Decomposition - The process of taking a global query and splitting it into the respective local queries.

DML - Data Manipulation/Management Language, language used by a DBMS to retrieve, delete, and update information in the database.

ECNDD - Extended Centralized Network Data Directory, a directory at every site in the DDBMS that contains a list of locations of data items that are updated from locations received from the CNDD.

Foreign Key - An attribute in a tuple for which it is not a primary key for that relation or record, but is for some other record or relation.

Generation - The process of creating a DML program, based upon certain inputs, that will duplicate the operation of the DML of a DBMS that uses another data model.

Global Schema - The relational representation of the integration of all of the Local Logical Schemas present in the DDBMS.

Heterogeneous DDBMS - A DDBMS composed of a collection of databases that use different data models.

Homogeneous DDBMS - A DDBMS composed of a collection of databases that use the same data model.

IMS - Information Managment System, a hierarchical database system developed by IBM.

Integration - The process of combining several local query results into a single result to be presented to the user.

LNDD - Local Network Data Directory, a directory at every DDBMS site that lists data items that are located on the host's database.

Local Logical Schema - The relational representation of the database present at the host system.

Partitioning - The division of common global data across several local databases.

Query - A question posed to the database system concerning the contents of that database.

Schema - A representation of the contents of a database. Another name for the intension of a database.

Subschema - A partial view of a database's contents.

TOTAL - A DDBMS marketed by Cincom Systems, Inc., based upon the network data model.

Translation - The process of converting the DML of one data model into that of another.

VAX-11/780 - A minicomputer manufactured by Digital Equipment Corporation.

Appendix B: TOTAL Data Management Language

(14:4.1 - 4.54)

Functional Description

The Data Management Language (DML) is a means of accessing and manipulating a defined data base. The language operates by invoking TOTAL through the CALL facility of the host programming language. When such a CALL is encountered, control is passed to TOTAL, which analyzes a parameter list to determine the function (i.e., "command") to be performed and the data to be acted upon. Communication between the application program and TOTAL is effected through work areas referenced in the parameter list. When control returns to the application program from TOTAL, a status code is also returned to indicate the result of the operation. If the operation completed successfully, a code of "*****" is returned. If the operation was unsuccessful, the data base is restored to its condition before the operation if necessary, and an appropriate status code is returned to indicate the cause of failure.

Command Parameters

As mentioned above, the parameter list in the CALL statement is the method of communication between TOTAL and the user's program. The parameters themselves are the names of areas defined elsewhere in the user's program. As might be expected of any called sub-program, TOTAL demands that the parameter list be in a certain order; the order shown must be strictly followed. Of the fifteen different parameters, some are used in every CALL to TOTAL, some depend on the particular type of

data set being accessed (i.e., master vs variable), and a few are used only in certain specialized functions. The following nine parameters fall into the first two categories, i.e., they are used in all but a few special functions:

OPERATION, STATUS, DATA-SET, REFERENCE,
LINKAGE-PATH, CONTROL-KEY, DATA-LIST,
DATA-AREA, END.

In usage they are best thought of as being grouped into three 'standard' parameter list formats.

PARAMETER	SERIAL FUNCTIONS	MASTER DATA-SET FUNCTIONS	VARIABLE DATA-SET FUNCTIONS
OPERATION	X	X	X
STATUS	X	X	X
DATA-SET	X	X	X
REFERENCE	X		X
LINKAGE-PATH			X
CONTROL-KEY		X	X
DATA-LIST	X	X	X
DATA-AREA	X	X	X
END.	X	X	X

In the descriptions and definitions which follow, certain notation conventions are used to express the format of a statement or a parameter. These are explained by the following rules.

1. Lower case letters are to be replaced by a symbol of the user's choosing.
2. Upper case letters are to be inserted as they appear.
3. Square brackets ([]) enclose a choice of options of which none, one, or several <u>may</u> be chosen.
4. Braces ({}) enclose a choice of options of which one and only one <u>must</u> be chosen.

The nine 'standard' parameters are described in detail below, before the discussion of the individual commands. There they will be shown where they occur, but described only to the extent that they vary from the discussion below. The only exception is the parameter OPERATION which will be shown as the operation code of the function to be performed.

NOTE: ALL PARAMETERS MUST BEGIN ON WORD BOUNDARIES

B.1 OPERATION:

This parameter is the name of ("points to") a five character field defined by the user into which he must place the operation code of the function to be performed, e.g., READM - read a master data set randomly.

B.2 STATUS:

This parameter is the name of ("points to") a four character field defined by the user into which TOTAL places a code indicating the result of the operation, e.g., "*****": the function has successfully completed; "FNTF": File Not Found and the function has not been performed. THIS FIELD SHOULD BE EXAMINED AFTER EVEPY COMMAND. A complete list of status codes and their meanings may be found in the Diagnostics Section.

B.3 DATA-SET:

This parameter is the name of ("points to") a four character field defined by the user into which the user must place the name of the data set to be operated upon as defined in a data base generation.

B.4 REFERENCE:

This parameter is the name of ("points to") a four character field defined by the user which is used to maintain the Internal Reference Point of the current variable record or a position in either a master of variable using the RDNXT function. This field is used by both TOTAL and the user to communicate information about processing along a relationship within a variable data set or along a specific role by inserting appropriate values into the reference field and expecting certain values to be present under

certain conditions. This may be best described by listing the acceptable contents of the reference field, qualified by the role of the participant.

B.4.1 LKxx

This is the last four characters of a linkage path name (mmmmLKxx) as defined in the Data Base Descriptor Module. The **user** places this value into the reference field to indicate that TOTAL is to retrieve a chain (depending on the operation code) and that processing is expected to continue along the specified linkage path. **TOTAL** places this value into the reference field to indicate that the first record of a chain has been deleted (this will be explained further below).

B.4.2 rrrr

This is the Internal Reference Point of the record currently being processed. The **user** places such a value into the reference field to directly retrieve a specific record whose Reference Point was previously known. The user also may place into the reference field a value which he previously saved upon interrupting continuous processing along a chain or reset a serial retrieval to some point in a data set. **TOTAL** places into the reference field the Internal Reference Point of the record just read, added, or written, or the "Back Pointer" from a deleted record (unless the deleted record was the first of a chain).

B.4.3 END.

This value is placed into the reference field by **TOTAL** when the user, while continuously processing along a chain of records, attempts to go beyond the end of the chain if reading forward or beyond the beginning if reading reverse. If this value is placed into the reference field by the **user** prior to execution of a TOTAL command, TOTAL will return a status code of "IRLC" or "IVRP".

B.4.4 BEGN

This value placed into the reference field by the user and used in conjunction with the 'RDNXT' function will cause the 'RDNXT' to start serially reading a specific data set at the absolute beginning of that file. Upon reaching the end of a file, 'END.' is placed in the reference by TOTAL.

The following table summarizes the effects of placing one of these values into the reference field prior to execution of a

TOTAL command. (This table is **not** intended to comprehensively describe the tabulated functions; a detailed explanation may be found in the list of commands.)

PROCESSING BASED ON THE CONTENT OF

REFERENCE FIELD BEFORE EXECUTION

	CONTENT		
FUNCTION	LKxx	rrrr	END.
READD	The operation is not performed, and a status code of IVRP is returned.	The record addressed by reference is retrieved.	IVRP
READR	The record at the end of the chain is retrieved.	The record logically before the one addressed by reference is retrieved.	IVRP
READV	The first record in the chain is retrieved.	The record logically after the one addressed by reference is retrieved.	IVRP

The following table shows the content of reference **after** execution of a TOTAL command.

CONTENT OF REFERENCE FIELD AFTER EXECUTION

FUNCTION	CONTENT
READD	Internal Reference Point of the record just read.
READR READV	Internal Reference Point of the record just read or 'END.' if the Read attempted to go off the end (or beginning) of the chain.

B.5 LINKAGE-PATH:

This parameter is the name of ("points to") an eight character field defined by the user into which he must place the eight character name of the linkage path (mmmmLKxx) as defined in the Data Base Descriptor Module. This is the vehicle through which the user dynamically names a specific relationship between a chain of variable records and a master record by the record control key. The terms "primary linkage path" and "controlling linkage path" refer to the linkage path named by the linkage-path parameter. The term "secondary linkage-path" refers to any othr linkage path defined for this record in the Data Base Descriptor.

B.6 CONTROL-KEY:

This parameter is the name of ("points to") a field defined by the user into which he places the record control key. TOTAL will "randomize" on this data, whether to locate a master record or to link from a master record to a variable record. If, during further processing of this command, it is found that the Control Key does not agree with the corresponding field in the user's data area, a status code of UCTL will be returned. To avoid this, it is recommended that the user name the control key field in the data area rather than define a separate field. The length of the Control Key is taken to be that defined in the Data Base Descriptor Module.

B.7 DATA-LIST:

This parameter is the name of ("points to") a list of data names. The list is a character string defined by the user which is composed of data names declared in the data base generation. This list must conform to the following format:

elem1elem2elem(n).....END.

The data names in the list may include:

- data elements
- data items
- control keys
- record codes

The list may **not** include:

- the ROOT field (master records only)
- linkpaths (variable records only)

The data names in the list may appear in any order and the data elements they name will be processed in the order listed. Thus the data list is ordered in the same manner as

the user's data area, not necessarily as the record on the data set.

Only the data elements named in the data list will be processed, i.e., transferred to or from the user's Data Area. It is suggested that the order of element names coincide with the generated order from DBGEN.

B.8 DATA-AREA:

This parameter is the name of ("points to") an area of memory defined by the user which is used as an input/output area for the data elements named in the Data List. The structure and characteristics of this area **must conform exactly** to the data elements as named in the Data List and in the same order.

B.9 END:

This parameter is the name of ("points to") a four character field defined by the user which must contain the value "END." or "RLSE". This parameter serves as a delimiter to the parameter list.

Description of DML Commands

The following pages list in alphabetic order all of the Data Management Language commands used by the translation program with a detailed description of each.

D.12 The Read Next Function

This function operates as a generalized serial retrieval method. The retrieval may be directed to a specific point in the dataset, namely, to the beginning or to a specific record location. Each record retrieved is placed in the Data Area and retrieval may continue by simply re-executing the command until the end of the dataset is reached. Only data records are returned; blank and control records are bypassed.

Required Parameters

RDNXT, STATUS, DATASET, QUALIFIER, DATA-LIST, DATA-AREA, END.

RDNXT: Mnemonic for Read Next

The user must place this mnemonic into the Operation Field.

QUALIFIER: Relative Record Number Field

This parameter is the name of a field defined by the user whichis used to maintain the current position in the dataset being processed. The content is always binary and four bytes in size. The field may contain:

BEGN: If the user places this into the Qualifier Field, then RDNXT retrieves the record physically first in the dataset and places it into the Data Area according to the Data List. The Internal Reference Point then replaces 'BEGN' and subsequent executions will then continue serially from that point.

rrrr: (Relative Record Number) The Internal Reference Point.

At the end-of-file, 'END.' is placed in the Qualifier Field and can optionally be placed in the Status Field by applying the appropriate patch.

D.14 The Read Master Function

This function operates by randomizing on the contents of the Control Key Field to find the specific record and place it into the Data Area according to the Data List.

Required Parameters

READM, STATUS, DATASET, CONTROL-KEY, DATA-LIST, DATA-AREA, END.

READM: Read Master Function Mnemonic

The user must place this mnemonic into the Operation Field.

STATUS: Status Code

Significant codes that may be returned are:

BCTL: Contol Key is null.
MRNF: Master Record not found.

D.16 The Read Variable Function

This function operates by logically following forward pointers along a specified linkage path. To read an entir hain, processing is initiated by placing LKxx into the Reference Field and issuing the READV command. TOTAL uses the Control Key to access a master record from which the pointer to the logical beginning of the chain is obtained. This first record of the chain is then returned to the user. Thereafter, processing continues by reissuing the READV command: since the Reference Field contains an Internal Reference Point, the forward chain is followed and records retrieved until the last record in the chain is reached. When this happens, TOTAL returns 'END.' in the Reference Field to indicate processing in complete.

Required Parameters

READV, STATUS, DATASET, REFERENCE, LINKAGE-PATH, CONTROL-KEY, DATA-LIST, DATA-AREA, END.

READV: Read Variable Function Mnemonic

The user must insert this mnemonic into the Operation Field.

STATUS: Status Code

Significant codes which may be returned are:

BCTL: Null Control Field
MLNF: Linkage Path is invalid for that file
IVRC: The record code in the path has not been defined
MRNF: The related Master record cannot be found
ICHN: The linkage path chain is invalid

REFERENCE: Internal Reference Point

If the Reference Field contains LKxx, the first logical record in the chain is retrieved. If the Reference Field contains an Internal Reference Point, TOTAL uses it to point to the record from which to obtain the forward pointer to the next record. When sucessfully completed, the Field contains the Internal

Reference Point of the record just read, or 'END.' when the logical end of the chain has been reached.

D.18 The Sign-Off Function

This function operates by physically closing any data sets which remain open and closing the log file.

Required Parameters

SINOF, STATUS, SCHEMA, END.

SINOF: Sign-Off Function Mnemonic

The user must place this mnemonic into the Operation Field.

STATUS: Status Code

Significant status codes which may be returned are:

LOAD: The file has exceeded its load limit.
FULL: The file is full and therefore cannot be added to.

Programming Considerations

1. All subsequent commands except a Sign-On will return a status code of "NOSO".
2. The SINOF is identical to the SINON except the function changes.
3. This should be the last statement, logically, in the user program prior to termination.
4. For an explanation of 'SCHEMA' refer to the 'SINON' function.

D.19 The Sign-On Function

This function must be the first CALL to the TOTAL system presented by the user program. This function operates in different

ways according to the mode of TOTAL being used (Singletask or Multitask). In a Singletask environment this function will initialize certain areas in TOTAL and the DBMOD which are linked to the user program, whereas in a Multitask environment this function will inform TOTAL that the application program will be perfomring communication as well as opening any files specified that have not previously been opened by another program.

This function allows the user to state the logging options desired, what datasets this program is to process, the mode of access for each dataset and the overall access provided to the task.

Required Parameters

SINON, STATUS, SCHEMA, END.

SINON: Sign-On Function Mnemonic

The user must place this mnemonic into the Operation Field.

STATUS: Status Code

Significant status codes which may be returned are:

EXSO: This sign-on was preceded by another sign-on without an interim SINOF.
DUPO: This file has already been opened. Fatal Condition.
FAIL: A communication error has occurred.
DBNF: The database stated has not been loaded by TOTAL.
LOCK: The file has not been recovered or is in use by another TOTAL program and cannot be used for update processing.
FULL: The file is full and therefore cannot be added to.
LOAD: The file has exceeded its load limit.
IBVF: The schema parameter is not terinated by "END." or, one or more of the subparameters within the schema is not the correct length. (e.g. one of the REALM entries is not 12 bytes in length.).

SCHEMA: Explicit Options and Files Needed for this Program Run

This parameter "points to" a field defined by the user in the following format and containing all below stated values:

1. Program Name - eight (8) character program name of this program

2. Data Base Descriptor Name - six (6) character DBMOD name

3. Access Mode - six (6) character field containing the general intention of this program

   RDONLY
   UPDATE
   RECOVR

   RECOVR: Entire set of functions available including WRITD if Singletask TOTAL is being used, or full variable serial read capability -- see RDNXT.

   RDONLY: Only read functions will be permitted.

   UPDATE: Entire set of DML functions available except WRITD.

4. Logging Options - two (2) character field identifying logging options

   NL: do not log
   LG: log before images only
   BI: log before images only
   FL: log functions only
   BF: log functions and before images

5. Realm - a group of twelve (12) character entries for each dataset in the data base required for this program and terminated by 'END.' literal.

   1. File name - four (4) character field containing a name of a dataset in the DBMOD

   2. File mode - four (4) character field containing the mode of file sharing needed. One entry for each data set required.

SHRE: This file may be shared among concurrent programs (Read Only).

PRIV: This file is exclusively assigned to this program which may have access to it during any program run. (UPDATE)

3. File Status - four (4) character field used for unique file status at OPEN time.

Programming Considerations

1. A Sign-On must be the first TOTAL command executed.

2. A second Sign-On may be issued after a Sign-Off, e.g., to change logging options, access mode, etc.

3. If any of the status fields used in the REALM entry are not "*****", then the general status will contain the proper error indicator. Checking of each REALM status is not required.

4. If any logging option other than 'NL' is used, then the desired DBMOD must have the appropriate log definitions as determined within the DBDL.

5. Logging is applied on a Data Base Descriptor rather than a task-by-task basis, and is only provided with TOTAL-Multitask.

Appendix C: Roth Relational System RETRIEVE Procedures

(15:122-124)

This information is included for purposes of increasing clarity of the example translations in the body of the thesis. No provisions currently exist for the actual translation of a Roth query into TOTAL DML, as no parsers or LNDD software are available.

Type "R" at the system level to enter RETRIEVE and the following prompt line is displayed:

Retrieve ops: G(et), S(ave), E(dit), X(ecute), D(isplay), Q(uit)

A concept central to the operation of RETRIEVE is the command file. A command file contains one or more relational queries. The command file can be created, modified, stored on disk, retrieved from disk, and executed. The commands which can reside in a command file are described below.

E. Join of two relations:

JOIN relation1, relation2 WHERE attr1 op attr2 GIVING relation3

where attr1 is in relation1 and attr2 is in relation2, op is =, <, >. The JOIN operation is a subset of the cartesian product where the condition of membership is specified in the WHERE clause. All restrictions under PRODUCT apply.

F. Project a relation over a subset of its attributes:

PROJECT relation1 OVER atrr1,attr2,...,attrN GIVING relation2

where attributes not specified in the OVER clause will be eliminated and any duplicate tuples will be eliminated. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

G. Select a subset of tuples from a relation:

SELECT ALL FROM relation1 WHERE condition GIVING relation2

where concition is a boolean predicate on the attributes of relation1 of the form a1 AND/OR a2 AND/OR a3 ..., where each aN is of the form attribute op value, where op is =, <, or >. The expression may be fully parenthesized to indicate the proper precedence of the operators, but ifnot, then AND has precedence over OR. One or more blanks or commas must be between each part ofthe command except that the left parenthesis may be flush against an item to its right, and the right parenthesis may be flush against an item to its left. Relation1 must have been attached and the READ password (if any) specified if the user does not own the relation, and relation2 must be unique.

Appendix D:

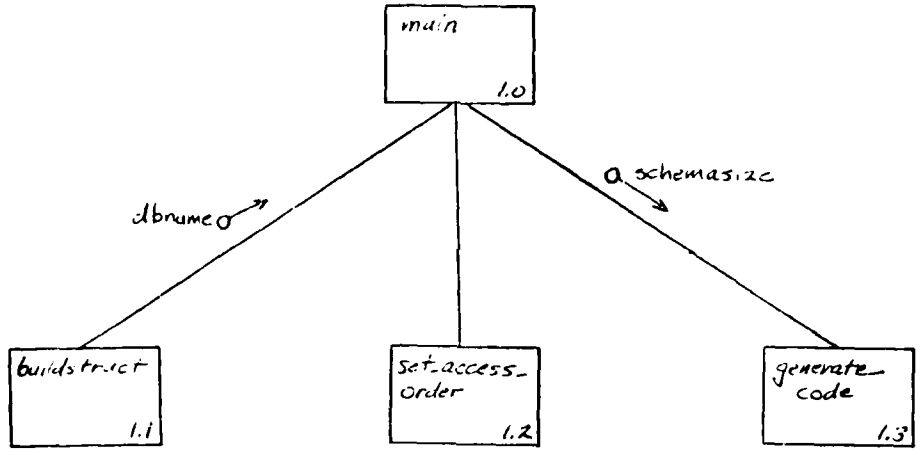
Structure Charts for

TRANS.C Translation Program

Structure Chart Index

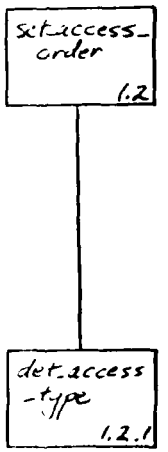
Chart	Module Number and Name
C1	1.0 main
	1.1 buildstruct
C2	1.2 set_access_order
	1.2.1 det_access_type
C3	1.3 generate_code
C4	1.3.1 decl_buftypes
	1.3.1.1 insert_node
	1.3.1.2 print_buffs
C5	1.3.2 decl_procedures
	1.3.2.1 readv
	1.3.2.2 readm
	1.3.2.3 rdnxt
	1.3.2.4 sonoroff
C6	1.3.3 totgen
C7	1.3.3.1 gen_qualifier
	1.3.3.1.1 gen_comparison
C6	1.3.3.2 outputall

AUTHOR: Capt Kevin Mahoney	DATE: 19 OCT 85	READER				
PROJECT: Relational to TOTAL Translator	REV: 2	DATE				



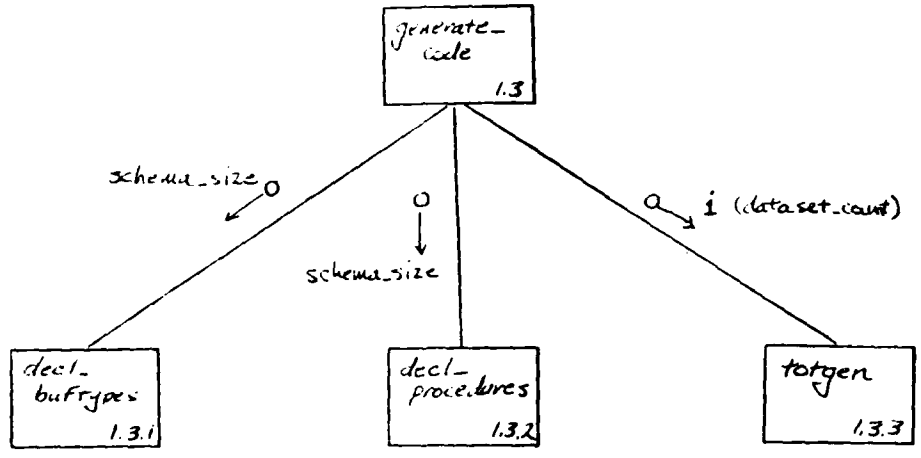
NODE: 1.0	TITLE: Main TRANSLATOR DRIVER	NUMBER: C1

AUTHOR: CAPT KEVIN MAHONEY	DATE: 19 OCT 85	READER				
PROJECT: RELATIONAL TO TOTAL TRANSLATION	REV: 2	DATE				



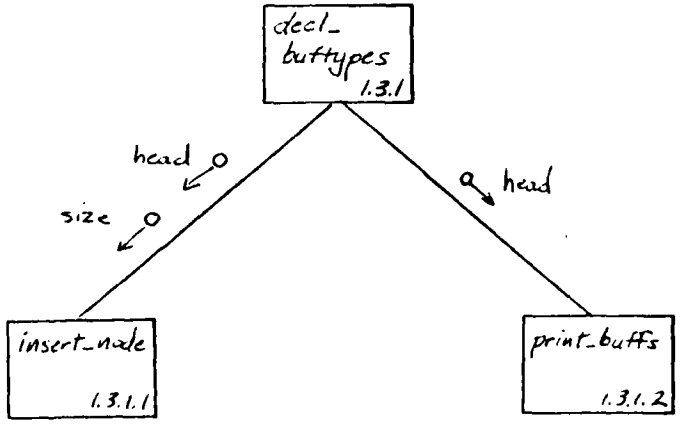
NODE: 1.2	TITLE: set_access_order	NUMBER: C2

AUTHOR: CAPT KEVIN MAHONEY	DATE: 19 OCT 85	READER				
PROJECT: RELATIONAL TO TOTAL TRANSLATOR	REV: 2	DATE				



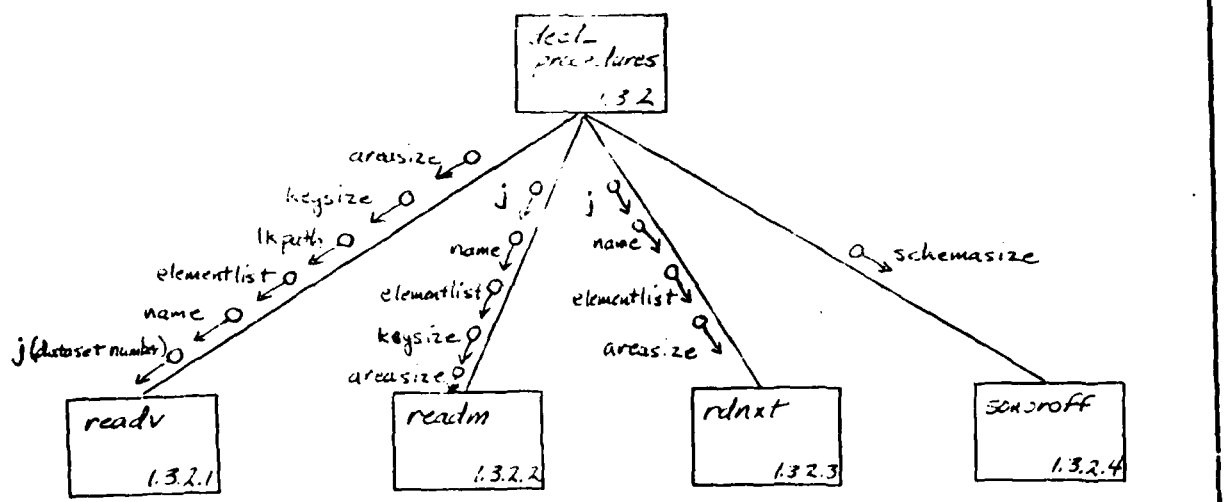
NODE: 1.3	TITLE: generate_code	NUMBER: C3

AUTHOR: Capt Kevin Mahoney	DATE: 19 Oct 85	READER				
PROJECT: Relational to Total Translator	REV: 2	DATE				



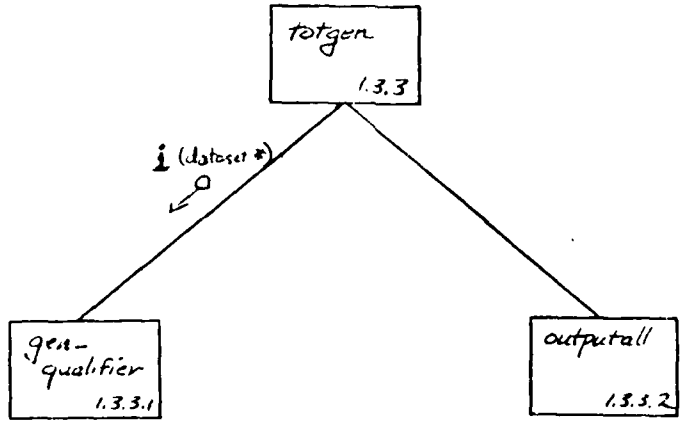
NODE: 1.3.1	TITLE: decl_buftypes	NUMBER: C4

AUTHOR: CAPT KEVIN MACKLEY	DATE: 19 OCT 85	READER				
PROJECT: RELATIONAL TO TOTAL TRANSLATOR	REV: 2	DATE				



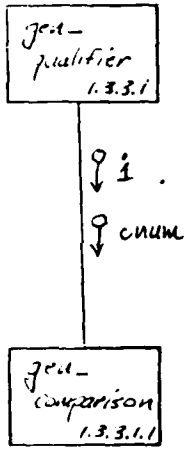
NODE: 1.3.2	TITLE: deel_procedures	NUMBER: C5

AUTHOR: CAPT KEVIN MAHONEY	DATE: 19 OCT 85	READER				
PROJECT: RELATIONAL TO TOTAL TRANSLATOR	REV: 2	DATE				



NODE: 1.3.3	TITLE: totgen	NUMBER: C6

AUTHOR: CAPT KEVIN MAHONEY	DATE: 19 OCT 85	READER				
PROJECT: RELATIONAL TO TOTAL TRANSLATOR	REV: 2	DATE				



NODE: 1.3.3.1	TITLE: gen-qualifier	NUMBER: C7

Appendix E:

Data Dictionary for

TRANS.C Structure Charts

Data Dictionary Entry for Process

NAME: main
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.0
DESCRIPTION: Main driver program for the translator
INPUT DATA: Database schema size
INPUT FLAGS: fopen (file open flag)
OUPUT DATA: None
OUTPUT FLAGS: None
GLOBAL DATA USED: Database schema file
GLOBAL DATA CHANGED: None
FILES READ: SCHEMA.DAT (database schema file)
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: None
PROCESSES CALLED: buildstruct
set_access_order
generate_code

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: buildstruct
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.1
DESCRIPTION: Reads in data directory information and builds an array of structures for use by the remainder of the program.
INPUT DATA: Query file
INPUT FLAGS: fopen (file open flag)
OUPUT DATA: None
OUTPUT FLAGS: None
GLOBAL DATA USED: dset - array of dataset structures
maxdset - number of datasets in query
GLOBAL DATA CHANGED: dset
maxdset
FILES READ: QUERY.DAT
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: main
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: set_access_order
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.2
DESCRIPTION: Orders the array of dataset structures into the most efficient processing sequence.
INPUT DATA: None
INPUT FLAGS: None
OUPUT DATA: None
OUTPUT FLAGS: None
GLOBAL DATA USED: dset array
GLOBAL DATA CHANGED: dset array
FILES READ: None
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: main
PROCESSES CALLED: det_access_type

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: det_access_type
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.2.1
DESCRIPTION: Determines the access type characteristics for each dataset in the array and assigns each of them a code specifying the particular access type. The types are (1) READM-file, (2) READM-literal, (3) RDNXT, and (4) READV.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: dset array
OUTPUT FLAGS: None
GLOBAL DATA USED: dset array
GLOBAL DATA CHANGED: dset array
FILES READ: None
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: set_access_order
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: generate_code
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3
DESCRIPTION: Driver for code generation portion of translator. It generates the beginning and ending sections of the translated program and calls subprocedures to do the rest.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: fopen (output file open flag)
GLOBAL DATA USED: dset array
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS (the generated translated program)
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: main
PROCESSES CALLED: decl_buftypes
decl_procedures
totgen

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: decl_buftypes
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1
DESCRIPTION: Generates buffer type declarations. Builds a linked list to eliminate duplicate buffer sizes before writing them.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: generate_code
PROCESSES CALLED: print_buffs
insert_node

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: insert_node
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1.1
DESCRIPTION: Inserts new buffer sizes into the linked list, eliminating duplicate size values.
INPUT DATA: buffer size (integer)
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_buftypes
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: print_buffs
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1.2
DESCRIPTION: Outputs the buffer declarations by following the linked list of buffer sizes.
INPUT DATA: linked list of integers
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_buftypes
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: decl_procedures
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.2
DESCRIPTION: Generates the subprocedure declarations within the generated program for the different calls to TOTAL.
INPUT DATA: dset array
schema length
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: generate_code
PROCESSES CALLED: readv
readm
rdnxt
sonoroff

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: readv
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.2.1
DESCRIPTION: Generates code for READ VARIABLE subprocedures.
INPUT DATA: dataset number, name, element list, buffer area size, linkpath name
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_procedures
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: readm
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.2.2
DESCRIPTION: Generates code for READ MASTER subprocedures.
INPUT DATA: dataset number, name, element list, buffer area size, key attribute size
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_procedures
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: rdnxt
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.2.3
DESCRIPTION: Generates code for READ NEXT MASTER subprocedures.
INPUT DATA: dataset number, name, element list, buffer area size
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_procedures
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: sonoroff
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.2.4
DESCRIPTION: Generates code for the SINON/SINOF subprocedure.
INPUT DATA: schema size, number of subschemas, schema file
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: "database-name"SC.DAT (schema file)
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_procedures
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: totgen
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.3
DESCRIPTION: Recursive module that generates the main body of translated code that determines the processing order of the query.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: generate_code
totgen
PROCESSES CALLED: totgen
gen_qualifier
outputall

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: gen_qualifier
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.3.1
DESCRIPTION: Generates the proper comparison code for single qualifiers or compound qualifiers seperated by a single AND or OR boolean conjunction/disjunction.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset array
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: totgen
PROCESSES CALLED: gen_comparison

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: gen_comparison
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.3.1.1
DESCRIPTION: Determines if comparison is on another dataset field or on a given literal and generates the appropriate code.
INPUT DATA: comparison number, dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset array
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: gen_qualifier
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: outputall
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.3.2
DESCRIPTION: Generates the code that will output the query results
when the translated program is executed.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: totgen
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: dbname
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Name of the database being queried
DATA TYPE: six (6) character string
MIN VALUE:
MAX VALUE:
RANGE:
VALUES:
STORAGE TYPE: process I/O
PART OF:
COMPOSITION:
ALIASES: None
PASSED FROM: buildstruct
PASSED TO: main

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: schemasize
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Size of the database schema in characters
DATA TYPE: Integer
MIN VALUE: 22
MAX VALUE: None
RANGE:
VALUES: 22 plus multiples of 12 (12, 24, 36, etc.)
STORAGE TYPE: file
PART OF:
COMPOSITION:
ALIASES: schema_size
PASSED FROM: main
PASSED TO: generate_code

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Alias

NAME: schema_size
TYPE: ALIAS
DD TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
SYNONYM: schemasize
PASSED FROM: generate_code
PASSED TO: decl_buftypes
decl_procedures
COMMENTS: None

Data Dictionary Entry for Alias

NAME: schema_size
TYPE: ALIAS
DD TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
SYNONYM: schemasize
PASSED FROM: decl_procedures
PASSED TO: sonoroff
COMMENTS: None

Data Dictionary Entry for Parameter

NAME: i
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Dataset number in the structure array (array subscript)
DATA TYPE: integer
MIN VALUE: 0
MAX VALUE: 7
RANGE:
VALUES:
STORAGE TYPE: Process I/O
PART OF:
COMPOSITION:
ALIASES: i
PASSED FROM: generate_code
PASSED TO: totgen

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Alias

NAME: i
TYPE: ALIAS
DD TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
SYNONYM: i
PASSED FROM: totgen
PASSED TO: gen_qualifier
COMMENTS: None

Data Dictionary Entry for Alias

NAME: i
TYPE: ALIAS
DD TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
SYNONYM: i
PASSED FROM: gen_qualifier
PASSED TO: gen_comparison
COMMENTS: None

Data Dictionary Entry for Parameter

NAME: head
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Pointer to the head of the linked list used to eliminate
duplicate buffer sizes.
DATA TYPE: Pointer
MIN VALUE:
MAX VALUE:
RANGE:
VALUES:
STORAGE TYPE: Process I/O
PART OF:
COMPOSITION:
ALIASES: head
PASSED FROM: decl_buftypes
PASSED TO: insert_node
print_buffs

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: size
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Size (in characters) of the particular field or literal
that requires a buffer type declaration.
DATA TYPE: integer
MIN VALUE: 1
MAX VALUE: None
RANGE: Positive integers
VALUES:
STORAGE TYPE: Process I/O
PART OF:
COMPOSITION:
ALIASES:
PASSED FROM: decl_buftypes
PASSED TO: insert_node

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: j
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Dataset number of the dataset being processed (dset array subscript).
DATA TYPE: integer
MIN VALUE: 0
MAX VALUE: 7
RANGE:
VALUES:
STORAGE TYPE: Process I/O
PART OF:
COMPOSITION:
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv
readm
rdnxt

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: name
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Name of the dataset being processed.
DATA TYPE: Four (4) character string
MIN VALUE:
MAX VALUE:
RANGE:
VALUES:
STORAGE TYPE: global
PART OF: dset structure array
COMPOSITION:
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv
readm
rdnxt

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: elementlist
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: This is a character string which is the concatenation of all the field names for the dataset being processed.
DATA TYPE: Character string terminated by "END."
MIN VALUE:
MAX VALUE:
RANGE:
VALUES: String must be multiple of eight (eight characters per field name), plus the four characters for the "END."
STORAGE TYPE: Process I/O
PART OF: Individual field names are part of dset array.
COMPOSITION: Field names and "END."
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv
readm
rdnxt

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: areasize
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Size required for the user area buffer where TOTAL places the retrieved data during a query. It is the sum of the sizes of all fields retrieved by a single call to TOTAL.
DATA TYPE: integer
MIN VALUE:
MAX VALUE:
RANGE: Positive integers
VALUES:
STORAGE TYPE: Process I/O
PART OF:
COMPOSITION:
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv
readm
rdnxt

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: keysize
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: Size of the dataset key that is needed for a READM or a READV call to TOTAL.
DATA TYPE: integer
MIN VALUE: 1
MAX VALUE:
RANGE: Positive integers
VALUES:
STORAGE TYPE: global
PART OF: dset array structure
COMPOSITION:
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv
readm

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: lkpath
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: The linkpath identifier for the particular dataset that requires a READV call to TOTAL.
DATA TYPE: Eight (8) character string
MIN VALUE:
MAX VALUE:
RANGE:
VALUES: The first four characters are the field name, the last four are "LKXX".
STORAGE TYPE: global
PART OF: dset array structure
COMPOSITION:
ALIASES:
PASSED FROM: decl_procedures
PASSED TO: readv

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Parameter

NAME: cnum
TYPE: PARAMETER
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
DESCRIPTION: The number of the qualifier being examined (1 or 2).
DATA TYPE: integer
MIN VALUE: 1
MAX VALUE: 2
RANGE:
VALUES:
STORAGE TYPE: global
PART OF: dset array structure (number of comparisions subscript)
COMPOSITION:
ALIASES:
PASSED FROM: gen_qualifier
PASSED TO: gen_comparison

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Appendix F: Configuration Management

for the TRANS.C Program

Required Files

There are three files in directory DUA3:[AFITDB.MAHONEY] on the Information Sciences Laboratory VAX-11/780 that are required for the translation process. These files are:

1. TRANS.C - source code of the translation program.
2. AFITDBSC.DAT - the input schema information file for the AFIT database. Additional databases will require their own schema files. Filenames are required to be the six (6) character database name (as defined in TOTAL) concatenated with "SC.DAT".
3. QUERY.DAT - the query input file. The format of the file is as shown on Page 87 of this thesis. Currently the user must create the file, but it will eventually be generated by the LNDD in response to a local query or subquery.

Instructions for Translating a Query

There are three steps in executing a translated query, execution of the translator, compilation and linking of the source code, and execution of the generated program. Specific details are as follows:

Translator Execution. After the required QUERY.DAT file has been created, enter "RUN TRANS" and the source code for the TOTAL query will be created in the file named TCODE.PAS.

Compilation. Enter "PASCAL TCODE" and the generated source code will be compiled.

Linking to TOTAL. The next task is to link the query to the TOTAL DBMS. For linking of a query against the AFIT database, enter the following statement:

```
LINK TCODE,DUA0:[AFITDB.TOTAL]NATDATBAS,NATBUF
```

This link statement is broken down as follows:

1. TCODE - the object code of the generated query program.
2. DUA0: - the prefix denoting the mass storage unit (disk) of the VAX that the TOTAL DBMS is resident on. If TOTAL is relocated on the system, this statement will be different.
3. [AFITDB.TOTAL] - directory and filename for the AFIT database. This will be different for other databases under TOTAL.
4. NATDATBAS,NATBUF - qualifier for the DATBAS FORTRAN interface program to TOTAL. This is unchanged for different databases.

Execution of Query. For actual execution of the query, enter "RUN TCODE" and the TOTAL database will be queried and the QRESULT.DAT file will be created. This query result file may then be displayed, printed or transferred back to the requesting node.

Changes to TRANS.C

If any changes are made to the source code of TRANS.C, the program must be recompiled and relinked. The steps are:

1. Enter "DEFINE LNK$LIBRARY SYS$LIBRARY:CRTLIB.OLB" to set the system C library as the link library for the program.
2. Enter "CC TRANS" to compile the source code.
3. Enter "LINK TRANS". The program is now ready for execution.

Appendix G:

TRANS.C Program Listings

```
/*****************************************************************
*                                                                *
*    DATE: 10/18/85                                              *
*    VERSION: 1.1                                                *
*    TITLE: Relational to TOTAL Query Translator                 *
*    FILENAME: trans.c                                           *
*    COORDINATOR: Capt Kevin H. Mahoney                          *
*    PROJECT: Thesis                                             *
*    OPERATING SYSTEM: VMS VERSION 4.2                           *
*    LANGUAGE: VAX-11 C                                          *
*    USE: Compile and link with standard library                 *
*    CONTENTS: main              - driver                        *
*              buildstruct       - input query information       *
*              set_access_order  - order query processing        *
*              generate_code     - code generation driver        *
*              decl_buftypes     - type declarations             *
*              insert_node       - eliminates duplicate sizes    *
*              print_buffs       - generates buffer types        *
*              decl_procedures   - procedures declaration driver *
*              readv             - read variable generator       *
*              readm             - read master generator         *
*              rdnxt             - read next master generator    *
*              sonoroff          - sign on/off TOTAL             *
*              totgen            - code body generator           *
*              gen_qualifier     - checks for con/disjunction    *
*              gen_comparison    - generates comparison code     *
*              outputall         - generates output code         *
*                                                                *
*    FUNCTION: The modules in this file take the query information*
*              (that the local network data directory (LNDD) sent *
*              in response to a global relational query) and     *
*              create a Pascal program with embedded calls to the *
*              TOTAL DBMS that produces results equivalent to    *
*              those requested by the original relational query. *
*              The first three routines read in the query infor- *
*              mation from the QUERY.DAT file, get the database  *
*              name from the query file, fill the dataset struc- *
*              ture array with information on each of the datasets*
*              in the query, and get the database schema from the *
*              <database>SC.DAT file.  The rest of the routines  *
*              utilize the array of dataset structues to create  *
*              the Pascal program under filename TCOD.PAS.  After *
*              TCODE.PAS has been created, it must be compiled and*
*              linked to TOTAL.  The command for linking to the  *
*              AFIT database on TOTAL is:                        *
*                                                                *
*              LINK TCODE,DUA0:[AFITDB.TOTAL]NATDATBAS,NATBUF    *
*                                                                *
*              It can then be executed, which places the results *
*              of the query into file QRESULT.DAT.               *
*                                                                *
*****************************************************************/
```

```
/*********************************************************************
*                                                                    *
*    DATE: 10/18/85                                                  *
*    VERSION: 1.1                                                    *
*    NAME: External Variable and Typedef Declarations                *
*                                                                    *
*    AUTHOR: Capt Kevin H. Mahoney                                   *
*                                                                    *
*    USE: These variables and types are external to the programs     *
*         in this file.  The dset structures and the dset array      *
*         size are heavily used throughout the modules, wiith        *
*         only the buildstruct module modifying the structures       *
*         (initially filling them from the query input file), so     *
*         extern was considered to be the best alternative.          *
*         Likewise, the external typedef of the linked list nodes    *
*         was chosen because their use was across three separate     *
*         modules.  The local database schema file and the           *
*         generated Pascal code file, tcode.pas, were also used      *
*         by several modules, so the pointers to these files were    *
*         also declared as extern.                                   *
*                                                                    *
*********************************************************************/

#include stdio

struct dataset {
   char name[5];          /* dataset name */
   char mory[2];          /* master/variable dataset indicator */
   char dskey[9];         /* data set key */
   int access_type;       /* readm,readv,rdnxt access indicator */
   char lkpth[9];         /* linkpath used - variable datasets */
   char lkref[5];         /* linkpath reference */
   int numflds;           /* number of fields requested */
   struct fld {
      char name[9];       /* field name */
      int size;           /* size of field */
      char outind[2];     /* indicates (Y or N) output requested */
      } field[40];        /* forty fields max for each dataset */
   int compnum;           /* number of comparisons on dataset */
   struct cmp {
      char cfld[9];       /* comparison field name */
      char op[3];         /* comparison operators <>,=,<,>,<=,>= */
      char comptype[2];   /* F for dataset field, L for literal */
      char argfld[9];     /* name of field used as qualifier */
      char arglit[60];    /* literal used as qualifier */
      char isandor[4];    /* AND, OR or XXX */
      } comp[2];          /* two qualifiers allowed per dataset */
   } dset[7];             /* seven dataset max in a query */

FILE *f1,*f3,*fopen();    /* tcode.pas and 'dbmname'sc.dat files */
int maxdset;              /* number of datasets in the query */
```

```
struct type_list {        /* linked list for buffer sizes */
   int data;
   struct type_list *next;
   };
typedef struct type_list NODE;
typedef NODE *LINK;
```

```
/*||1.0*/
/*************************************************************************
*                                                                        *
*    DATE: 10/14/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: main                                                          *
*    MODULE NUMBER: 1.0                                                  *
*    DESCRIPTION: main driver program for the translator                 *
*    PASSED VARIABLES: N/A                                               *
*    RETURNS: N/A                                                        *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: schema.dat                                              *
*    FILES WRITTEN: None                                                 *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: buildstruct - input query, place in structures      *
*                    set_access_order - optimize access order            *
*                    generate_code - generate Pascal program             *
*    CALLING MODULES: None                                               *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/10/85) Original Version                            *
*             1.1 (10/14/85) Changed parameter passed to generate_       *
*                    code from num_subschemas to schemasize and          *
*                    parameter passed to buildstruct to db_name.         *
*                                                                        *
*************************************************************************/

main()
   {
   int schemasize;
   extern FILE *f3,*fopen;
   char db_name[13];       /* database schema file */

   /* get the database name and build the dataset structures */
   buildstruct(db_name);
   /* order the structures */
   set_access_order();

   /* open the proper database schema file */
   strcat(db_name,"SC.DAT");
   if ((f3 = fopen(db_name,"r")) == NULL) {
      printf("Can't open the schema file.\n");
      exit(1);
      }

   fscanf(f3,"%d",&schemasize);
   generate_code(schemasize);
   fclose(f3);
} /* end of main */
```

```
/*|||1.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/14/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: buildstruct                                                   *
*    MODULE NUMBER: 1.1                                                  *
*    DESCRIPTION: This module reads in the data directory                *
*                 information that has been placed in the QUERY.DAT      *
*                 file by the DDBMS and inserts the information into     *
*                 the 'dset' array of structures for further             *
*                 processing by the access ordering and code             *
*                 generating procedures.                                 *
*    PASSED VARIABLES: dbname - name of the TOTAL database accessed      *
*    RETURNS: dbname                                                     *
*    GLOBAL VARIABLES USED: dset - array of dataset structures           *
*                           maxdset - number of datasets in query        *
*    GLOBAL VARIABLES CHANGED: dset, maxdset                             *
*    FILES READ: query.dat                                               *
*    FILES WRITTEN: None                                                 *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: main                                               *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*             1.1 (10/10/85) Moved fopen() to main, added check for      *
*                   argument type (field or literal) in comparison       *
*             1.2 (10/14/85) Made dbname a parameter to the module       *
*                   instead of extern.                                   *
*                                                                        *
*************************************************************************/

buildstruct(dbname)

   char dbname[];

   {
   FILE *f2,*fopen();
   int i,j,temp;
   extern struct dataset dset[];
   extern int maxdset;

   if ((f2 = fopen("query.dat","r")) == NULL) {
      printf("Can't open the query file.\n");
      exit(1);
      }

   /* get the databse name and number of datasets in query */
   fscanf(f2,"%s",dbname);
   fscanf(f2,"%d",&maxdset);
```

```
   for (i=0; i<maxdset; i++)
      {
      fscanf(f2,"%s",(dset[i].name));   /* dataset information */
      fscanf(f2,"%s",(dset[i].morv));
      fscanf(f2,"%s",(dset[i].dskey));
      if (!stremp((dset[i].morv),"V"))
         {
         fscanf(f2,"%s",(dset[i].lkpth)); /* variable dataset info */
         fscanf(f2,"%s",(dset[i].lkref));
         }
      fscanf(f2,"%d",&temp);
      dset[i].numflds = temp;
      for (j=0; j<(dset[i].numflds); j++)
         {
         fscanf(f2,"%s",(dset[i].field[j].name)); /* element info */
         fscanf(f2,"%d",&temp);
         dset[i].field[j].size = temp;
         fscanf(f2,"%s",(dset[i].field[j].outind));
         }
      fscanf(f2,"%d",&temp);
      dset[i].compnum = temp;
      for (j=0; j<(dset[i].compnum); j++)
         {
         fscanf(f2,"%s",(dset[i].comp[j].cfld)); /* qualifiers */
         fscanf(f2,"%s",(dset[i].comp[j].op));
         fscanf(f2,"%s",(dset[i].comp[j].comptype));
         if (!stremp((dset[i].comp[j].comptype),"F"))
             fscanf(f2,"%s",(dset[i].comp[j].argfld));
         else
             fscanf(f2,"%s",(dset[i].comp[j].arglit));

         fscanf(f2,"%s",(dset[i].comp[j].isandor));
         }
      }  /* end for - all datasets in the query have been read in */

   fclose(f2);
}  /* end buildstruct */
```

```
/*|1.2*/
/*************************************************************************
*                                                                        *
*    DATE: 10/18/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: set_access_order                                              *
*    MODULE NUMBER: 1.2                                                  *
*    DESCRIPTION:  This module takes the array of datset structures      *
*                  and places them in the most efficient access          *
*                  order for the code generation procedure.              *
*    PASSED VARIABLES: None                                              *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dataset array (dset)                         *
*    GLOBAL VARIABLES CHANGED: dataset array (dset)                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: None                                                 *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: det_access_type                                     *
*    CALLING MODULES: main                                               *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/10/85) Program Stub                                *
*             1.1 (10/18/85) Added call to det_access_type               *
*                                                                        *
*************************************************************************/

set_access_order()

   {       /* program stub except for call to det_access_type */
   det_access_type();
   return;
   }
```

```
/*|||1.2.1*/
/************************************************************************
*                                                                       *
*    DATE: 10/18/85                                                     *
*    VERSION: 1.0                                                       *
*    NAME: det_access_type                                              *
*    MODULE NUMBER: 1.2.1                                               *
*    DESCRIPTION: This module take the array of datasset structures     *
*              and determines what the access type for each will be.    *
*              It then assigns the dset.access_type field the proper    *
*              indicator value.  The values and their meanings are:     *
*              1 - READM (Read Unique Master) with a previously         *
*                 retrieved dataset field as the key qualifier.         *
*              2 - READM with a given input literal (from the user)     *
*                 as the qualifier on the dataset key.                  *
*              3 - RDNXT (Read Sequential Master) for both previous     *
*                 fields and literals as qualifying arguments.          *
*              4 - READV (read Variable) for variable datasets.         *
*    PASSED VARIABLES: None                                             *
*    RETURNS: None                                                      *
*    GLOBAL VARIABLES USED: dataset array (dset)                        *
*    GLOBAL VARIABLES CHANGED: dataset array (dset)                     *
*    FILES READ: None                                                   *
*    FILES WRITTEN: None                                                *
*    HARDWARE INPUT: N/A                                                *
*    HARDWARE OUTPUT: N/A                                               *
*    MODULES CALLED: None                                               *
*    CALLING MODULES: set_access_order                                  *
*                                                                       *
*    AUTHOR: Capt Kevin H. Mahoney                                      *
*    HISTORY: 1.0 (10/18/85) Original Version                           *
*                                                                       *
************************************************************************/

det_access_type()

   {
   extern struct dataset dset[];
   extern int maxdset;
   int i, literalsize;

   for (i=o; i<maxdset; i++)
      {
      if (!strcmp((dset[i].morv),"M"))
         /* master dataset */

         if (!strcmp(dset[i].dskey),(dset[i].comp[0].cfld)))
            /* comparison is on a dataset key field */

            if (!strcmp((dset[i].comp[0].op),"="))
               /* equality comparison */
```

Data Dictionary Entry for Process

NAME: generate_code
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3
DESCRIPTION: Driver for code generation portion of translator. It generates the beginning and ending sections of the translated program and calls subprocedures to do the rest.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: fopen (output file open flag)
GLOBAL DATA USED: dset array
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS (the generated translated program)
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: main
PROCESSES CALLED: decl_buftypes
decl_procedures
totgen

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: decl_buftypes
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1
DESCRIPTION: Generates buffer type declarations. Builds a linked list to eliminate duplicate buffer sizes before writing them.
INPUT DATA: dset array
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: dset
maxdset
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: generate_code
PROCESSES CALLED: print_buffs
insert_node

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: insert_node
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1.1
DESCRIPTION: Inserts new buffer sizes into the linked list,
eliminating duplicate size values.
INPUT DATA: buffer size (integer)
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: None
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_buftypes
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

Data Dictionary Entry for Process

NAME: print_buffs
TYPE: PROCESS
PROJECT: Relational to TOTAL DML Translation Program (Thesis)
NUMBER: 1.3.1.2
DESCRIPTION: Outputs the buffer declarations by following the linked list of buffer sizes.
INPUT DATA: linked list of integers
INPUT FLAGS: None
OUPUT DATA: Pascal code statements (text)
OUTPUT FLAGS: None
GLOBAL DATA USED: None
GLOBAL DATA CHANGED: None
FILES READ: None
FILES WRITTEN: TCODE.PAS
HARDWARE READ: None
HARDWARE WRITTEN: None
ALIASES: None
CALLING PROCESSES: decl_buftypes
PROCESSES CALLED: None

VERSION: 1.0
DATE: 10 November 1985
AUTHOR: Capt Kevin H. Mahoney

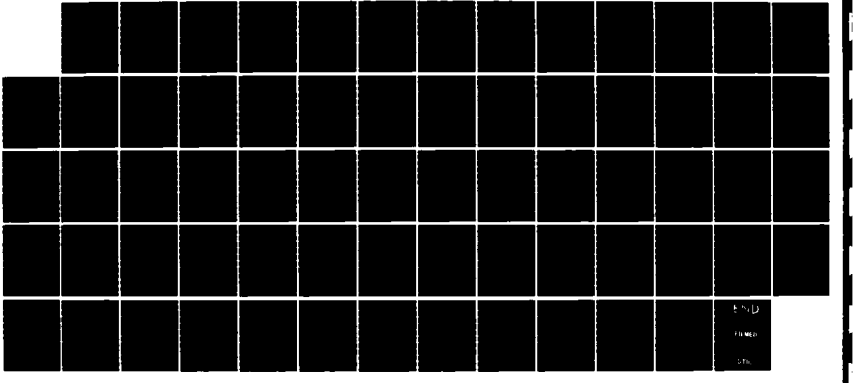
```
                if (!strcmp((dset[i].comp[0].comptype),"F"))
                   /* comparison is on a previous field */
                   dset[i].access_type = 1;

                else
                   /* comparison is on a literal */
                   {
                   literalsize = strlen(dset[i].comp[0].arglit);

                   if (literalsize == (dset[i].field[0].size))
                      /* sizes match exactly */
                      dset[i].access_type = 2;
                   else
                      /* size doesn't match, RDNXT needed */
                      dset[i].access_type = 3;
                   }
             else
                /* was not an equality comparison */
                dset[i].access_type = 3;

          else
             /* was not on a dataset key field */
             dset[i].access_type = 3;

       else
          /* is a variable dataaset, not a master */
          dset[i].access_type = 4;
       }

} /* end det_access_type */
```

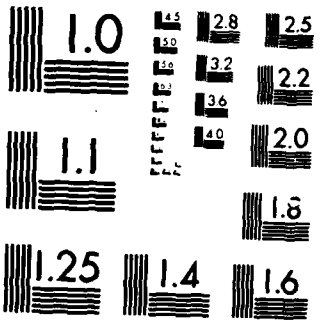
```
/*|||1.3.*/
/*************************************************************************
*                                                                        *
*    DATE: 10/14/85                                                      *
*    VERSION: 1.4                                                        *
*    NAME: generate_code                                                 *
*    MODULE NUMBER: 1.3                                                  *
*    DESCRIPTION: This is the main module for the generation of the      *
*                 Pascal code that will actually execute the query.      *
*                 It generates the beginning and ending statements       *
*                 of the program, and calls subprocedures to do the      *
*                 remaining code generation.                             *
*    PASSED VARIABLES: schema_size - size of the schema file             *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dataset structure array (dset)               *
*                           maxdset - number of datasets in query        *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: decl_buftypes - generate buffertype declaration     *
*                    decl_procedures - generate subprocedure decls.      *
*                    totgen - generate the main body of Pascal code      *
*    CALLING MODULES: main                                               *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/5/85) Original Version was main()                  *
*             1.1 (10/10/85) Renamed as generate_code when new main      *
*                 was written, moved procedure declarations to a         *
*                 separate module, added call to decl_buftypes.          *
*             1.2 (10/11/85) Added test and generation for literals      *
*                 shorter than the test field, creating TEMPnn           *
*             1.3 (10/13/85) Added code for generation of indexes        *
*                 for decomposition of user areas in the query           *
*             1.4 (10/14/85) Made schema_size the parameter              *
*                 replacing numschm, number of subschemas in file        *
*                                                                        *
*************************************************************************/

generate_code(schema_size)

   int schema_size;

   {
   extern struct dataset dset[];
   extern FILE *f1, *fopen();
   extern int maxdset;
   int i,j,k, areasize, literalsize, index_max;

   /*open the generated code file */
   if ((f1 = fopen("tcode.pas","w")) == NULL) {
```

AD-A164 013 THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR..(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOOL OF ENGI.. K H MAHONEY

3/3

UNCLASSIFIED DEC 85 AFIT/GCS/ENG/85D-7 F/G 9/2 NL





```
   printf("Can't open the generated file.\n");
   exit(1);
   }

/* print the opening statements of the Pascal program */
fprintf(f1,"PROGRAM GENTCODE(INPUT,QRESULT);\n);

/* declare labels */
fprintf(f1,"LABEL ");
for (j=0; j<(maxdset*3); j++)
   fprintf(f1,"%d",j);
fprintf(f1,"ERRL;\n\n");

/* declare type definitions of buffers */
decl_buftypes(schema_size);

/* declare structure types */
for (j=0; j<maxdset; j++)
   {
   fprintf(f1,"\nSTRUC%d = RECORD;\n",j);
   for (k=0; k<(dset[j].numflds); k++)
      fprintf(f1,"  A%d : BUFF%d;\n",k,(dset[j].field[k].size));
   fprintf(f1,"END;\n");
   }  /* end for */

/* declare variables */
fprintf(f1,"\nVAR\n");
fprintf(f1,"  STATUS : BUFF4;\n");
fprintf(f1,"  FCTN : BUFF5;\n");
fprintf(f1,"  INDEX : INTEGER;\n");
fprintf(f1,"  NOTFOUND : BOOLEAN;\n");
fprintf(f1,"  QRESULT : TEXT;\n");
index_max = 0;

for (j=0; j<maxdset; j++)
   {
   fprintf(f1,"  S%d : STRUC%d;\n",j,j);

   /* this code keeps track of the maximum number of indices */
   /* that will be needed to decompose the TOTAL user area   */
   /* into the dataset structer fields for output, etc...    */
   if (index_max < (dset[j].numflds))
      index_max = (dset[j].numflds);

   /* generate user area sizes and key (if needed) sizes */
   areasize = 0;
   for (k=0; k<(dset[j].numflds); k++)
      {
      /* see if a requested field is a dataset key */
      if (!strcmp((dset[j].dskey),(dset[j].field[k].name)))
         fprintf(f1,"  KEY%d : BUFF%d;\n",
               j,(dset[j].field[k].size));
```

```
            areasize = areasize + (dset[j].field[k].size);
            }
         fprintf(f1,"  UAREA%d : BUFF%d;\n",j,areasize);

         /* declare TEMP holders in case comp literal is shorter */
         for (k=0; k<(dset[j].compnum); k++)
            {
            if (!strcmp((dset[j].comp[k].comptype),"L"))  /* literal */
               {
               literalsize = strlen(dset[j].comp[k].arglit);
               fprintf(f1,"  TEMP%d%d : BUFF%d;\n",j,k,literalsize);
               }
            }

         if (!strcmp((dset[j].morv),"V"))
            fprintf(f1,"  REF%d : BUFF4;\n",j);
         else
            fprintf(f1,"  QUAL%d : BUFF4;\n",j);
         } /* end for */

      /* declare counter indexes for decomposing the user area */
      for (j=0; j<index_max; j++)
         fprintf(f1,"  IND%d : INTEGER;\n,j);

      /* declare all subprocedures for TOTAL calls */
      decl_procedures(schema_size);

      fprintf(f1,"    BEGIN\n");
      fprintf(f1,"      REWRITE(QRESULT);\n");
      fprintf(f1,"      FCTN := 'SINON';\n");
      fprintf(f1,"      SONOROFF(FCTN);\n");
      fprintf(f1,"      IF STATUS <> '****' THEN GOTO ERRL;\n");

      totgen(0);  /* create the main body of the Pascal Program */

      /* print the finishing statements of the Pascal program */
      fprintf(f1,"ERRL: FCTN := 'SINOF';\n");
      fprintf(f1,"      SONOROFF(FCTN);\n");
      fprintf(f1,"      CLOSE(QRESULT);\n");
      fprintf(f1,"END.\n");
      fclose(f1);

   }  /* end generate_code */
```

```
/*|||1.3.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/13/85                                                      *
*    VERSION: 1.2                                                        *
*    NAME: decl_buftypes                                                 *
*    MODULE NUMBER: 1.3.1                                                *
*    DESCRIPTION: Build a linked list of buffer sizes in order to        *
*                 eliminate duplicates.  The BUFF type declarations      *
*                 are then output to tcode.pas                           *
*    PASSED VARIABLES: schemalength - size of the schema file            *
*    RETURNS: N/A                                                        *
*    GLOBAL VARIABLES USED: dset array of structures, maxdset            *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: print_buffs - print buffer type declarations        *
*                    insert_node - insert a new size into the list       *
*    CALLING MODULES: generate_code                                      *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*             1.1 (10/10/85) Changed name of module from build_buf       *
*             1.2 (10/13/85) Added code for computing the user area      *
*                    size and changed parameter to schemalength          *
*                                                                        *
*************************************************************************/

decl_buftypes(schemalength)

   int schemalength;
   {

   extern struct dataset dset[];
   extern FILE *f1;
   extern int maxdset;
   LINK head, htemp;
   int i,j,temp,areasize;

   /* initialize the buffer size list with known values */
   head = (LINK) malloc(sizeof(NODE));
   head->data = 1;
   head->next = (LINK) malloc(sizeof(NODE));
   head->next->data = 2;
   head->next->next = (LINK) malloc(sizeof(NODE));
   head->next->next->data = 3;
   head->next->next->next = (LINK) malloc(sizeof(NODE));
   htemp = head->next->next->next;
   htemp->data = 4;
   htemp->next = (LINK) malloc(sizeof(NODE));
```

```
        htemp->next->data = 5;
        htemp->next->next = (LINK) malloc(sizeof(NODE));
        htemp->next->next->data = 8;
        htemp->next->next->next = NULL;

        /* insert schema size */
        insert_node(head,schemalength);

        /* process the datasets and insert values */
        for (i=0; i<maxdset; i++)
           {
           areasize = 0;
           /* figure the element list size and insert it */
           temp = ((dset[i].numflds)*8)+4;
           insert_node(head,temp);

           for (j=0; j<(dset[i].numflds); j++)
              {
              temp = (dset[i].field[j].size);
              areasize = areasize + temp;     /* figuring the user area */
                                              /* size for the call to TOTAL */
              insert_node(head,temp);
              }
           /* insert user area size */
           insert_node(head,areasize);

           for (j=0; j<(dset[i].compnum); j++)
              {
              if (!stremp((dset[i].comp[j].comptype),"L"))
                 {
                 temp = strlen(dset[i].comp[j].arglit);
                 insert_node(head,temp);
                 }
              }
           }

        /* list is finished, so now print declarations */
        fprintf(f1, "TYPE\n");
        print_buffs(head);

     } /* end of decl_buftypes */
```

```
/*|||1.3.1.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/7/85                                                       *
*    VERSION: 1.0                                                        *
*    NAME: insert_node                                                   *
*    MODULE NUMBER: 1.3.1.1                                              *
*    DESCRIPTION: inserts new values into the buffer size list,          *
*                 eliminating duplicate size values                      *
*    PASSED VARIABLES: head, size                                        *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: None                                                 *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_buftypes                                      *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*                                                                        *
*************************************************************************/

insert_node(head,size)

   LINK head;
   int size;

   {
   LINK pointer, temp;
   int flag;

   /* there is no null list case - 1,2,3,4,5, and 8 are in list */
   if (size>(head->data)) /* eliminates zero or negative sizes */
      {
      flag = 1;
      pointer = head;
      while ((flag!=0) && (pointer->next!=NULL))
         {
         if ((pointer->next->data) > size)
            {
            flag = 0;
            temp = pointer->next;
            pointer->next = (LINK) malloc(sizeof(NODE));
            pointer->next->data = size;
            pointer->next->next = temp;
            }
         else if ((pointer->next->data) == size)
            flag = 0;
         else
```

```
            pointer = pointer->next;
            }
         if (flag==1)  /* new end of list case */
            {
            pointer->next = (LINK) malloc(sizeof(NODE));
            pointer->next->data = size;
            pointer->next->next = NULL;
            }
         }
   }  /* end of insert_node */
```

```
/*||1.3.1.2*/
/*************************************************************************
*                                                                        *
*    DATE: 10/7/85                                                       *
*    VERSION: 1.0                                                        *
*    NAME: print_buffs                                                   *
*    MODULE NUMBER: 1.3.1.2                                              *
*    DESCRIPTION: This module uses the linked list of buffer sizes       *
*                 to print out the buffer type declarations in the       *
*                 generated Pascal file by calling itself recursively    *
*    PASSED VARIABLES: head                                              *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_buftypes                                      *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*                                                                        *
*************************************************************************/

print_buffs(head)

   LINK head;

   {
   extern FILE *f1;
   if (head!=NULL)
      {
      fprintf(f1,"  BUFF%d = PACKED ARRAY",head->data);
      fprintf(f1,"[1..%d] OF CHAR;\n",head->data);
      print_buffs(head->next);
      }

}  /* end of print_buffs */
```

```
/*|||1.3.2*/
/*************************************************************************
*                                                                        *
*    DATE: 10/18/85                                                      *
*    VERSION: 1.2                                                        *
*    NAME: decl_procedures                                               *
*    MODULE NUMBER: 1.3.2                                                *
*    DESCRIPTION: This module generates subprocedure declarations        *
*                 within the Pascal program for the different calls      *
*    PASSED VARIABLES: schmlen - size of schema file                     *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dataset structure array (dset)               *
*                           maxdset - number of datasets in query        *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: readv - 'read variable dataset' subprocedure        *
*                    readm - 'read master direct' subprocedure           *
*                    rdnxt - 'read next master' subprocedure             *
*                    sonoroff - 'sign on or off TOTAL' subprocedure      *
*    CALLING MODULES: generate_code                                      *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/10/85) Original Version created from code          *
*                    originally in generate_code                         *
*             1.1 (10/14/85) Made schmlen the passed parameter           *
*             1.2 (10/18/85) Changed the strcmp checks for the           *
*                    dataset access characteristics to a simple          *
*                    comparison on dset[i].access_type field.            *
*                                                                        *
*************************************************************************/

decl_procedures(schmlen)

   int schmlen;

   {
   int j,k,keysize,areasize;
   extern struct dataset dset[];
   extern int maxdset;
   extern FILE *f1;
   char elementlist[300];

   for (j=0; j<maxdset; j++)
      {
      areasize = 0;
      strcpy(elementlist,"\0");  /* blank out the element list */

      /* build the list of required fields and their total length */
      for (k=0; k<(dset[j].numflds); k++)
```

```
            {
            strcat(elementlist,(dset[j].field[k].name));
            areasize = areasize + (dset[j].field[k].size);
            }

         /* check for dataset type */
         if (dset[j].access_type == 4)  /* this is a variable dataset */
            {
            /* set the keysize to the previous dataset control field */
            /* size because this is the master record that the       */
            /* variable dataset chain is accessed from               */
            keysize = (dset[j-1].field[0].size);
            readv(j,(dset[j].name),elementlist,
                  (dset[j].lkpth),keysize,areasize);
            }
         else  /* master dataset */
            {
            if (dset[j].access_type == 3)
               /* must read master dataset sequentially */
               rdnxt(j,(dset[j].name),elementlist,areasize);
            else
               {
               /* read unique master, type 1 or 2 */
               keysize = (dset[j].field[0].size);
               readm(j,(dset[j].name),elementlist,keysize,areasize);
               }
            }
         }
      /* generate the sign on/off procedure */
      sonoroff(schmlen);

   } /* end of decl_procedures */
```

```
/*|||1.3.2.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/11/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: readv                                                         *
*    MODULE NUMBER: 1.3.2.1                                              *
*    DESCRIPTION: This module creates the Pascal code for each           *
*                 declaration of a READ VARIABLE subprocedure within*
*                 the generated program.                                 *
*    PASSED VARIABLES: j - the dataset number                            *
*                      name - the dataset name                           *
*                      elementlist - requested dataset field names       *
*                      areasize - buffer size to retrieve field          *
*                      keysize - size of the control key field           *
*                      lkpth - linkpath name to variable                 *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_procedures                                    *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*             1.1 (10/11/85) Inserted code for handling element          *
*                    lists longer than 40 characters.                    *
*                                                                        *
*************************************************************************/

readv(j,name,elementlist,lkpth,keysize,areasize)

   int j,keysize,areasize;
   char name[];
   char elementlist[];
   char lkpth[];

   {
   int listsize,loops,leftover,i,k;
   extern FILE *f1;

   strcat(elementlist,"END.");
   listsize = strlen(elementlist);
   loops = listsize/40;
   leftover = listsize%40;
   fprintf(f1,"\nPROCEDURE READV%d(VAR REF:BUFF4; CTRLKEY:BUFF%d; ",
         j,keysize);
   fprintf(f1,"VAR AREA:BUFF%d);\n",areasize);
   fprintf(f1,"  VAR\n");
```

```
    fprintf(f1,"    FUNCT : BUFF5;\n");
    fprintf(f1,"    DSET  : BUFF4;\n");
    fprintf(f1,"    ENDP  : BUFF4;\n");
    fprintf(f1,"    ELIST : BUFF%d;\n",listsize);
    fprintf(f1,"    LKPTH : BUFF8;\n");
    fprintf(f1,"\n  PROCEDURE DATBAS(%cSTDESCR FUNCT:BUFF5; ",'%');
    fprintf(f1,"STATUS:BUFF4;\n          DSET:BUFF4; REF:BUFF4; ");
    fprintf(f1,"LKPTH:BUFF8;\n          CTRLKEY:BUFF%d; ",keysize);
    fprintf(f1,"ELIST:BUFF%d; AREA:BUFF%d;\n",listsize,areasize);
    fprintf(f1,"          ENDP:BUFF4); FORTRAN;\n");
    fprintf(f1,"  BEGIN\n");
    fprintf(f1,"    FUNCT := 'READV';\n");
    fprintf(f1,"    STATUS := '    ';\n");
    fprintf(f1,"    ENDP := 'END.';\n");
    fprintf(f1,"    DSET := '%s';\n",name);

    /* print out all of the fields that make up the element list */
    fprintf(f1,"    ELIST := '");
    for (i=0; i<loops; i++)
       {
       if (i>0)
          fprintf(f1,"\n           + '");
       for (k=0; k<40; k++)
          fprintf(f1,"%c",elementlist[k+(i*40)]);
       fprintf(f1,"'");
       }
    if (loops>0)
       fprintf(f1,"\n           + '");
    for (i=0; i<leftover; i++)
       fprintf(f1,"%c",elementlist[i+(loops*40)]);
    fprintf(f1,"';\n");

    fprintf(f1,"    LKPTH := '%s';\n",lkpth);
    fprintf(f1,"    FOR INDEX := 1 TO %d DO\n",areasize);
    fprintf(f1,"       AREA[INDEX] := ' ';\n");
    fprintf(f1,"    DATBAS(FUNCT,STATUS,DSET,REF,LKPTH,CTRLKEY,");
    fprintf(f1,"ELIST,AREA,ENDP);\n");
    fprintf(f1,"  END;\n\n");

} /* end readv */
```

```
/*|||1.3.2.2*/
/*************************************************************************
*                                                                        *
*    DATE: 10/11/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: readm                                                         *
*    MODULE NUMBER: 1.3.2.2                                              *
*    DESCRIPTION: This module creates the Pascal code for each           *
*                 declaration of a READ MASTER subprocedure within       *
*                 the generated program.                                 *
*    PASSED VARIABLES: j - the dataset number                            *
*                      name - the dataset name                           *
*                      elementlist - requested dataset field names       *
*                      areasize - buffer size to retrieve field          *
*                      keysize - size of the control key field           *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_procedures                                    *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*             1.1 (10/11/85) Inserted code for handling element          *
*                  lists longer than 40 characters.                      *
*                                                                        *
*************************************************************************/

readm(j,name,elementlist,keysize,areasize)

   int j,keysize,areasize;
   char name[];
   char elementlist[];

   {
   int listsize,loops,leftover,i,k;
   extern FILE *f1;

   strcat(elementlist,"END.");
   listsize = strlen(elementlist);
   loops = listsize/40;
   leftover = listsize%40;
   fprintf(f1,"\nPROCEDURE READM%d(CTRLKEY:BUFF%d; ",j,keysize);
   fprintf(f1,"VAR AREA:BUFF%d);\n",areasize);
   fprintf(f1,"  VAR\n");
   fprintf(f1,"    FUNCT : BUFF5;\n");
   fprintf(f1,"    DSET  : BUFF4;\n");
   fprintf(f1,"    ENDP  : BUFF4;\n");
```

```
    fprintf(f1,"    ELIST : BUFF%d;\n",listsize);
    fprintf(f1,"\n  PROCEDURE DATBAS(%cSTDESCR FUNCT:BUFF5; ",'%');
    fprintf(f1,"STATUS:BUFF4;\n          DSET:BUFF4; REF:BUFF4; ");
    fprintf(f1,"CTRLKEY:BUFF%d; ELIST:BUFF%d;\n",keysize,listsize);
    fprintf(f1,"      AREA:BUFF%d; ENDP:BUFF4); FORTRAN;\n",areasize);
    fprintf(f1,"  BEGIN\n");
    fprintf(f1,"    FUNCT := 'READM';\n");
    fprintf(f1,"    STATUS := '    ';\n");
    fprintf(f1,"    ENDP := 'END.';\n");
    fprintf(f1,"    DSET := '%s';\n",name);

    /* print out all of the fields that make up the element list */
    fprintf(f1,"    ELIST := '");
    for (i=0; i<loops; i++)
       {
       if (i>0)
          fprintf(f1,"\n           + '");
       for (k=0; k<40; k++)
          fprintf(f1,"%c",elementlist[k+(i*40)]);
       fprintf(f1,"'");
       }
    if (loops>0)
       fprintf(f1,"\n           + '");
    for (i=0; i<leftover; i++)
       fprintf(f1,"%c",elementlist[i+(loops*40)]);
    fprintf(f1,"';\n");

    fprintf(f1,"    FOR INDEX := 1 TO %d DO\n",areasize);
    fprintf(f1,"       AREA[INDEX] := ' ';\n");
    fprintf(f1,"    DATBAS(FUNCT,STATUS,DSET,CTRLKEY,");
    fprintf(f1,"ELIST,AREA,ENDP);\n");
    fprintf(f1,"  END;\n\n");

} /* end readm */
```

```
/*|||1.3.2.3*/
/*************************************************************************
*                                                                        *
*    DATE: 10/11/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: rdnxt                                                         *
*    MODULE NUMBER: 1.3.2.3                                              *
*    DESCRIPTION: This module creates the Pascal code for each           *
*                 declaration of a sequential READ MASTER                *
*                 subprocedure within the generated program.             *
*    PASSED VARIABLES: j - the dataset number                            *
*                      name - the dataset name                           *
*                      elementlist - requested dataset field names       *
*                      areasize - buffer size to retrieve field          *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_procedures                                    *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/7/85) Original Version                             *
*             1.1 (10/11/85) Inserted code for handling element          *
*                   lists longer than 40 characters.                     *
*                                                                        *
*************************************************************************/


rdnxt(j,name,elementlist,areasize)

   int j,areasize;
   char name[];
   char elementlist[];

   {
   int listsize,loops,leftover,i,k;
   extern FILE *f1;

   strcat(elementlist,"END.");
   listsize = strlen(elementlist);
   loops = listsize/40;
   leftover = listsize%40;
   fprintf(f1,"\nPROCEDURE RDNXT%d(VAR QUAL:BUFF4; ",j);
   fprintf(f1,"VAR AREA:BUFF%d);\n",areasize);
   fprintf(f1,"  VAR\n");
   fprintf(f1,"    FUNCT : BUFF5;\n");
   fprintf(f1,"    DSET  : BUFF4;\n");
   fprintf(f1,"    ENDP  : BUFF4;\n");
```

```
    fprintf(f1,"    ELIST : BUFF%d;\n",listsize);
    fprintf(f1,"\n  PROCEDURE DATBAS(%cSTDESCR FUNCT:BUFF5; ",'%');
    fprintf(f1,"STATUS:BUFF4;\n          DSET:BUFF4; REF:BUFF4; ");
    fprintf(f1,"CTRLKEY:BUFF%d; ELIST:BUFF%d;\n",keysize,listsize);
    fprintf(f1,"      AREA:BUFF%d; ENDP:BUFF4); FORTRAN;\n",areasize);
    fprintf(f1,"  BEGIN\n");
    fprintf(f1,"    FUNCT := 'RDNXT';\n");
    fprintf(f1,"    STATUS := '    ';\n");
    fprintf(f1,"    ENDP := 'END.';\n");
    fprintf(f1,"    DSET := '%s';\n",name);

    /* print out all of the fields that make up the element list */
    fprintf(f1,"    ELIST := '");
    for (i=0; i<loops; i++)
       {
       if (i>0)
          fprintf(f1,"\n           + '");
       for (k=0; k<40; k++)
          fprintf(f1,"%c",elementlist[k+(i*40)]);
       fprintf(f1,"'");
       }
    if (loops>0)
       fprintf(f1,"\n           + '");
    for (i=0; i<leftover; i++)
       fprintf(f1,"%c",elementlist[i+(loops*40)]);
    fprintf(f1,"';\n");

    fprintf(f1,"    FOR INDEX := 1 TO %d DO\n",areasize);
    fprintf(f1,"       AREA[INDEX] := ' ';\n");
    fprintf(f1,"    DATBAS(FUNCT,STATUS,DSET,QUAL,");
    fprintf(f1,"ELIST,AREA,ENDP);\n");
    fprintf(f1,"  END;\n\n");

} /* end rdnxt */
```

```
/*|||1.3.2.4*/
/*************************************************************************
*                                                                        *
*    DATE: 10/14/85                                                      *
*    VERSION: 1.2                                                        *
*    NAME: sonoroff                                                      *
*    MODULE NUMBER: 1.3.2.4                                              *
*    DESCRIPTION: This module creates the Pascal code for each           *
*                 declaration of the SINON or SINOF to TOTAL             *
*                 subprocedure within the generated program.             *
*    PASSED VARIABLES: schemasize - the number of characters in the      *
*                      schema definition file.                           *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: None                                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: "databasename"sc.dat                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: decl_procedures                                    *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/5/85) Original Version                             *
*             1.1 (10/10/85) Changed parameter from schema_size to       *
*                    num_nubs, added 'hold' string, added loop for       *
*                    splitting schema into subschema strings             *
*             1.2 (10/14/85) Changed parameter back to schemasize        *
*                    and added a fscanf to get the number of sub-        *
*                    schemas that are in the database schema file.       *
*                                                                        *
*************************************************************************/

sonoroff(schemasize)

   int schemasize;

   {
   int i,num_subs;
   extern FILE *f1;
   char hold[50];
   extern FILE *f3;

   fscanf(f3,"%d",&num_subs);
   fprintf(f1,"\nPROCEDURE SONOROFF(FUNCT:BUFF5);\n");
   fprintf(f1,"  VAR\n");
   fprintf(f1,"    ENDP  : BUFF4;\n");
   fprintf(f1,"    SCHEMA : BUFF%d;\n",schemasize);
   fprintf(f1,"\n  PROCEDURE DATBAS(%cSTDESCR FUNCT:BUFF5; ",'%');
   fprintf(f1,"STATUS:BUFF4;\n          ");
   fprintf(f1,"SCHEMA:BUFF%d; ENDP:BUFF4); FORTRAN;\n",schemasize);
   fprintf(f1,"  BEGIN\n");
```

```
        fprintf(f1,"    STATUS := '    ';\n");
        fprintf(f1,"    ENDP := 'END.';\n");
        for (i=0; i<num_subs; i++)
           {
           /* get next subschema string */
           fscanf(f3,"%s",hold);
           if (i==0)  /* first string */
              fprintf(f1,"    SCHEMA := '%s'",hold);
           else
              fprintf(f1,"\n         + '%s'",hold);
           }
        fprintf(f1,";\n");  /* end of subschema string assignment */
        fprintf(f1,"    DATBAS(FUNCT,STATUS,SCHEMA,ENDP);\n");
        fprintf(f1,"  END;\n\n");
     }  /* end sonoroff */
```

```
/*|||1.3.3*/
/*************************************************************************
*                                                                        *
*    DATE: 10/18/85                                                      *
*    VERSION: 1.4                                                        *
*    NAME: totgen                                                        *
*    MODULE NUMBER: 1.3.3                                                *
*    DESCRIPTION: Recursive subprocedure that generates the main         *
*                 body of Pascal code that determines the order of       *
*                 calls to the TOTAL database.                           *
*    PASSED VARIABLES: i - the level of recursion                        *
*    RETURNS: N/A                                                        *
*    GLOBAL VARIABLES USED: dset structure array                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: totgen - calls itself recursively for each dset*
*                    gen_qualifier - generates proper qualifier code*
*                     for up to two comparisons with AND or OR    *
*                    outputall - generates "output information" code*
*                       when innermost loop has been executed.      *
*    CALLING MODULES: generate_code, totgen                              *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/5/85) Original Version                             *
*             1.1 (10/10/85) Created the gen_comparison module from *
*                    duplicated code in totgen                           *
*             1.2 (10/14/85) Output format changes                       *
*             1.3 (10/17/85) Added code for determining whether the *
*                    dataset key argument is a literal or if it is  *
*                    a previously retrieved dataset field value.    *
*                    Also moved comparison qualifier code to the new*
*                    routine gen_qualifier and to gen_comparison.   *
*             1.4 (10/18/85) Corrected errors in the conditionals    *
*                    that determine access type of the dataset.     *
*                    In the process, changed stremp comparisons     *
*                    into comparisons on dset[i].access_type field. *
*                                                                        *
*************************************************************************/

totgen(i)

   int i;

   {
   extern struct dataset dset[];
   extern FILE *f1;
   extern int maxdset;
   int j,k,n,offset,found;
```

```
if (i >= maxdset)
   {
   outputall();  /* print out all of the fields */
   return;
   }
if (dset[i].access_type < 4)  /* master dataset */
   {
   if ((dset[i].access_type == 1)||(dset[i].access_type == 2))
      {
      /* comparison against a dataset key */

      if (dset[i].access_type == 1)  /* the argument is a field */
         {
         found = 0; /* initialize flag */
         for (k=o; k<i; k++)
            {
            /* search all previously retrieved datasets */

            for (n=0; n<(dset[k].numflds); n++)
               {
               /* check all fields of each previous dataset */
               if (!strcmp((dset[k].field[n].name),
                     (dset[i].comp[0].argfld)));
                  {
                  /* previous field matching argument found */
                  fprintf(f1,"       KEY%d := S%d.A%d;\n",i,k,n,);
                  found = 1;
                  break;  /* exit 'n' loop since found */
                  }
               }
            if (found==1)
               break;    /* exit 'k' loop since found */
            }
         }
      else   /* comparison argument is a literal */
         fprintf(f1,"       KEY%d := '%s';\n",
                  i,(dset[i].comp[0].arglit));

      /* this code is common to both type 1 and 2 access */

      fprintf(f1,"     READM%d(KEY%d,UAREA%d);\n",i,i,i);
      fprintf(f1,"     IF STATUS = 'MRNF' THEN GOTO %d;\n",(3*i));
      fprintf(f1,"     IF STATUS <> '****' THEN GOTO ERRL;\n");
      fprintf(f1,"        WITH S%d DO BEGIN\n",i);
      offset = 0;
      for (j=0; j<(dset[i].numflds); j++)
         {
         fprintf(f1,"          FOR IND%d := 1 TO %d DO\n",
                  j,(dset[i].field[j].size));
         fprintf(f1,"       A%d[IND%d] := UAREA%d[IND%d + %d];\n",
                  j,j,i,j,offset);
         offset = offset + (dset[i].field[j].size);
```

```
            }
         fprintf(f1,"     END;\n\n");

         /* generate qualifier code */
         if (dset[i].compnum > 0)
            gen_qualifier(i);

         /* generate the inner loop for the next dataset */
         totgen(i+1);
         fprintf(f1,"%d: ;\n",(3*i));  /* loop ending label */

         } /* end unique master dataset key retrieval */

      else  /* sequential search of master dataset is needed */

         {
         fprintf(f1,"     QUAL%d := 'BEGN';\n",i);
         fprintf(f1,"     RDNXT%d(QUAL%d,UAREA%d);\n",i,i,i);
         fprintf(f1,"%d:   IF QUAL%d = 'END.' THEN GOTO %d;\n",
                    (3*i+2),i,(3*i));
         fprintf(f1,"     IF STATUS <> '****'THEN GOTO ERRL;\n);
         fprintf(f1,"\n     WITH S%d DO BEGIN\n",i);
         offset = 0;
         for (j=0; j<(dset[i].numflds); j++)
            {
            fprintf(f1,"       FOR IND%d := 1 TO %dDO\n",
                    j,(dset[i].field[j].size));
            fprintf(f1,"       A%d[IND%d] := UAREA%d[IND%d + %d];\n",
                    j,j,i,j,offset);
            offset = offset + (dset[i].field[j].size);
            }
         fprintf(f1,"      END;\n\n");

         /* generate qualifier code */
         if (dset[i].compnum > 0)
            gen_qualifier(i);

         totgen(i+1);

         fprintf(f1,"%d:  RDNXT%d(QUAL%d,UAREA%d);\n",(3*i+1),i,i,i);
         fprintf(f1,"    GOTO %d;\n",(3*i+2));
         fprintf(f1,"%d:  ;\n",(3*i));
         }
      }  /* end master dataset */

   else  /* variable dataset */

      {
      fprintf(f1,"    REF%d := '%s';\n",i,(dset[i].lkref));
      fprintf(f1,"    READV%d(REF%d,S%d.A0,UAREA%d);\n",i,i,i,(i-1),i);
      fprintf(f1,"%d:  IF REAF%d = 'END' THEN GOTO %d;\n",
                 (3*i+2),i,(3*i));
```

```
        fprintf(f1,"    IF STATUS <> '****' THEN GOTO ERRL;\n");
        fprintf(f1,"\n    WITH S%d DO BEGIN\n",i);
        offset = 0;

        for (j=0; j<(dset[i].numflds); j++)
           {
           fprintf(f1,"       FOR IND%d := 1 TO %dDO\n",
                    j,(dset[i].field[j].size));
           fprintf(f1,"       A%d[IND%d] := UAREA%d[IND%d + %d];\n",
                    j,j,i,j,offset);
           offset = offset + (dset[i].field[j].size);
           }
        fprintf(f1,"      END;\n\n");

        /* generate qualifier code */
        if (dset[i].compnum > 0)
           gen_qualifier(i);

        totgen(i+1);

        fprintf(f1,"%d:     READV%d(REF%d,S%d.A0,UAREA%d);\n",
                 (3*i+1),i,i,(i-1),i);
        fprintf(f1,"    GOTO %d;\n",(3*i+2));
           fprintf(f1,"%d:  ;\n",(3*i));

        }  /* end variable dataset */
    }  /* end totgen */
```

```
/*|||1.3.3.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/17/85                                                      *
*    VERSION: 1.0                                                        *
*    NAME: gen_qualifier                                                 *
*    MODULE NUMBER: 1.3.3.1                                              *
*    DESCRIPTION: This module checks to see if the qualifying            *
*                 comparison on the dataset is a single or compound      *
*                 boolean.  Only one conjunction/disjunction is          *
*                 allowed for the dataset.  It then calls                *
*                 gen_comparison to generate the proper code for         *
*                 the type of comparison that is made on the field.      *
*    PASSED VARIABLES: i - current datset comparison is for              *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dset structure array                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: gen_comparison                                      *
*    CALLING MODULES: totgen                                             *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/17/85) Original Version - created from code        *
*                    originally in totgen and gen_comparison plus        *
*                    code to check for ANDs and ORs.                     *
*                                                                        *
*************************************************************************/

gen_qualifier(i)

   int i;

   {
   char conjunct[4];
   int j,n,lit_length;
   extern FILE *f1;
   extern struct dataset dset[];

   for (n=0; n<(dset[i].compnum; n++)
      {
      /* check for literals, because temp holders must come first */
      if (!strcmp((dset[i].comp[n].comptype),"L"))
         {
         lit_length = strlen(dset[i].comp[n].arglit);
         for (j=0; j<(dset[i].numflds; j++)
            {
            /* see which field the literal matches up to */
            if (!strcmp((dset[i].field[j].name),
                  (dset[i].comp[n].cfld)))
```

```
                {
                /* found, so generate literal assignment code */
                fprintf(f1,"  FOR INDEX := 1 TO %d DO\n",lit_length);
                fprintf(f1,"    TEMP%d%d[INDEX] := S%d.A%d[INDEX];\n",
                        i,n,i,j);
                }
             }
          }
       } /* end 'j' loop */

    /* now check to see if there is an AND or OR */
    /* and generate the appropriate code         */

    strcpy(conjunct,(dset[i].comp[0].isandor));
    if ((!strcmp(conjunct,"AND"))||(!strcmp(conjunct,"OR")))
       {
       /* two qualifying comparisons on the field */
       fprintf(f1,"      IF (");
       gen_comparison(i,0);  /* generate the first comparison */
       fprintf(f1,") %s \n",conjunct);
       fprintf(f1,"         (");
       gen_comparison(i,1);  /* generate the second comparison */
       fprintf(f1,") THEN\n");
       }
    else  /* single comparison */
       {
       fprintf(f1,"      IF ");
       gen_comparison(i,0);
       fprintf(f1," THEN\n");
       }
    fprintf(f1,"       NOTFOUND := FALSE\n");
    fprintf(f1,"     ELSE NOTFOUND := TRUE;\n");

    if ((dset[i].access_type <= 2))
       /* READM with extra qualifier */
       fprintf(f1,"    IF NOTFOUND THEN GOTO %d;\n",(3*i));
    else
       /* sequential or variable read */
       fprintf(f1,"    IF NOTFOUND THEN GOTO %d;\n",(3*i+1));

} /* end gen_qualifier */
```

```
/*|||1.3.3.1.1*/
/*************************************************************************
*                                                                        *
*    DATE: 10/17/85                                                      *
*    VERSION: 1.1                                                        *
*    NAME: gen_comparison                                                *
*    MODULE NUMBER: 1.3.3.1.1                                            *
*    DESCRIPTION: This module generates the proper code for the          *
*                 type of comparison that is made on the qualifying      *
*                 field, whether it is against another field from a      *
*                 previously retrieved dataset, or against a given       *
*                 literal, possibly shorter than the field checked.      *
*    PASSED VARIABLES: i - current dataset comparison is for             *
*                      cnum - current comparison number (1st or 2nd)     *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dset structure array                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: gen_qualifier                                      *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/11/85) Original Version - created from code        *
*                   in totgen plus additional tests for comparison       *
*                   type.                                                *
*             1.1 (10/17/85) Removed some selection code to gen_         *
*                   qualifier and moved more comparison checking         *
*                   in from totgen.  All temporary literal assign-       *
*                   ment code was removed to gen_qualifier.              *
*                                                                        *
*************************************************************************/

gen_comparison(i,cnum)

   int i,cnum;

   {
   int j,k,n,found;

   for (j=0; j<(dset[i].numflds); j++)
      {
      /* first see which field the comparison is on */
      if (!strcmp((dset[i].field[j].name),(dset[i].comp[cnum].cfld)))

         /* field found, generate comparison code */
         if (!strcmp((dset[i].comp[cnum].comptype),"F"))
            {
            /* comparison is against a previously retrieved field */
            found = 0;
```

```
                for (k=0; k<i; k++)
                   {
                   /* search all previously retrieved datasets */
                   for (n=0; n<(dset[k].numflds); n++)
                      {
                      /* check all fields of each previous dataset */
                      if (!strcmp((dset[k].field[n].name),
                                 (dset[i].comp[cnum].argfld)))
                         {
                         /* previous field that matches has been found */
                         fprintf(f1,"S%d.A%d %s S%d.A%d",i,j,
                                 (dset[i].comp[cnum].op),k,n);
                         found = 1;
                         break;   /* found, so exit 'n' loop */
                         }
                      }
                   if (found==1)
                      break;      /* found, so exit 'k' loop */
                   }
                }

             else  /* comparison is on a given literal */

                fprintf(f1,"TEMP%d%d %s '%s'",i,cnum,
                    (dset[i].comp[cnum].op),(dset[i].comp[cnum].arglit));
          }
       } /* end gen_comparison */
```

```
/*|||1.3.3.2*/
/*************************************************************************
*                                                                        *
*    DATE: 10/9/85                                                       *
*    VERSION: 1.0                                                        *
*    NAME: outputall                                                     *
*    MODULE NUMBER: 1.3.3.2                                              *
*    DESCRIPTION: This module is triggered when the recursion level      *
*                 of TOTGEN goes beyond the number of datasets to        *
*                 be retrieved.  It then generates the Pascal code       *
*                 that will print out the retrieved values to the        *
*                 query result file.                                     *
*    PASSED VARIABLES: None                                              *
*    RETURNS: None                                                       *
*    GLOBAL VARIABLES USED: dset structure array                         *
*    GLOBAL VARIABLES CHANGED: None                                      *
*    FILES READ: None                                                    *
*    FILES WRITTEN: tcode.pas                                            *
*    HARDWARE INPUT: N/A                                                 *
*    HARDWARE OUTPUT: N/A                                                *
*    MODULES CALLED: None                                                *
*    CALLING MODULES: totgen                                             *
*                                                                        *
*    AUTHOR: Capt Kevin H. Mahoney                                       *
*    HISTORY: 1.0 (10/9/85) Original Version                             *
*                                                                        *
*************************************************************************/

outputall()

   {
   extern FILE *f1;
   extern int maxdset;
   int i,j,linelength;

   for (i=0; i<maxdset; i++)  /* check all datasets */
      {
      linelength = 0;
      fprintf(f1,"     WRITELN(QRESULT");
      for (j=0; j<(dset[i].numflds); j++)  /* check all fields */
         {
         if (!stremp((dset[i].field[j].outind),"Y"))
            {
            /* field is requested for output */
            if ((linelength + (dset[i].field[j].size)) < 80)
               {
               linelength = linelength + (dset[i].field[j].size) + 5;
               fprintf(f1,"S%d.A%d,\n",i,j);
               fprintf(f1,"            '    '"); /* output spacer */
               }
            else
               {
```

```
                fprintf(f1,");\n");
                fprintf(f1,"      WRITELN (QRESULT");
                fprintf(f1,"S%d.A%d,\n",i,j);
                fprintf(f1,"            '    '"); /* output spacer */
                linelength = linelength + (dset[i].field[j].size) + 5;
                }
            }
        }
      fprintf(f1,");\n");
      }
   fprintf(f1,"    WRITELN(QRESULT);\n\n");
} /* end outputall */
```

Appendix H:

Test Query Input and Results

Test Query Number One

Roth Query

```
SELECT ALL FROM Student WHERE (Name = 'Mahoney')
     GIVING Temp1
JOIN Temp1, Enrolled-In WHERE (Temp1.SSAN = Enrolled-In.SSAN)
     GIVING Temp2
SELECT ALL FROM Temp2 WHERE (Temp2.Quarter-Year = 'FA85') OR
     (Temp2.Quarter-Year = 'WI85') GIVING Temp3
JOIN Temp3, Course WHERE (Temp3.Number = Course.Number)
     GIVING Temp4
PROJECT Temp4 OVER (Student.SSAN, Student.Name, Course.Number,
     Course.Title) GIVING Temp5
```

Input File	Result File
AFITDB	062084021 MAHONEY, KEVIN H.
3	EENG793 FA84
STDT	ADVANCED SOFTWARE ENG
M	WAIV
STDTCTRL	
2	062084021 MAHONEY, KEVIN H.
STDTCTRL	EENG588 FA84
9	COMPUTER SYSTEMS ARCHITECT
Y	T WAIV
STDTNAME	
28	062084021 MAHONEY, KEVIN H.
Y	EENG589 FA84
1	OPER SYS & FILE STRUCTURES
STDTNAME	S WAIV
=	
L	062084021 MAHONEY, KEVIN H.
MAHONEY	EENG587 FA84
XXX	MINICOMPUTER/MICROPROC LAB
VCQR	B WAIV
V	
VCQRCODE	062084021 MAHONEY, KEVIN H.
STDTLKCQ	EENG698 WI85
LKCQ	THESIS SEMINAR
4	WAIV
VCQRCODE	
2	062084021 MAHONEY, KEVIN H.
N	EENG646 WI85
VCQRNMBR	COMPUTER DATA BASE SYS
8	WAIV
Y	
VCQRIDEN	062084021 MAHONEY, KEVIN H.

Test Query Number One (continued)

Input File

```
4
Y
VCQRGRAD
2
Y
VCQRIDEN
=
L
FA84
OR
VCQRIDEN
=
L
WI85
XXX
MCRS
M
MCRSCTRL
2
MCRSCTRL
8
N
MCRSTITL
50
Y
1
MCRSCTRL
F
VCQRNMBR
XXX
```

Result File

```
EENG685       WI85
  ADVANCED ALGORITHM DESIGN
  WAIV

062084021      MAHONEY, KEVIN H.
MATH666       WI85
  PERSPECTIVES IN PROG LANGU
U WAIV

(8 TUPLES IN ALL)
```

Test Query Number Two

Roth Query

```
SELECT ALL FROM Student WHERE (Student.Name > 'C') AND
     (Student.Name < 'G') GIVING Temp1
PROJECT Temp1 OVER (Student.Rank, Student.Name)
     GIVING Temp2
```

Input File

```
AFITDB
1
STDT
M
STDTCTRL
2
STDTRANK
3
Y
STDTNAME
28
Y
2
STDTNAME
>
L
C
AND
<
L
G
XXX
```

Result File

```
2LT     Dxxxxxxx, ZEKI
MAJ     Exxxxxx, YASER ALY
CPT     Fxxxxx, PAUL G.
1LT     Dxxx, JUAN E.
CPT     Fxxxxxxxxx, PHILIP B.
1LT     Dxxxxx, WILLIAM M.
1LT     Exxxx, STEVEN P.
2LT     Exxxxxxx, ROBERT A
2LT     Dxxxxx, PAUL J.
CPT     Dxxxxxx, PETER W.
1LT     Fxxxxxx, ROY A.
2LT     Fxxxxx, LARRY E.
1ST     Dxxxxxx, FRANK W.
CPT     Exxxxxx, CARLOS R.
-
-
-
1LT     Fxxxxxx, MARK L.
(37 TUPLES IN ALL)
```

(Last names were deleted for privacy purposes)

Test Query Number Three

Roth Query

```
SELECT ALL FROM Student WHERE (Student.SSAN = '062084021')
     GIVING Temp1
PROJECT Temp1 OVER (Student.Rank, Student.Name)
     GIVING Temp2
```

Input File

```
AFITDB
1
STDT
M
STDTCTRL
3
STDTCTRL
9
N
STDTRANK
3
Y
STDTNAME
28
Y
1
STDTCTRL
=
L
062084021
XXX
```

Result File

```
LT      MAHONEY, KEVIN H.
```

Test Query Number Four

Roth Query

```
SELECT ALL FROM Course WHERE (Course.Number = 'EENG')
     GIVING Temp1
PROJECT Temp1 OVER (Course.Number, Course.Title)
     GIVING Temp2
```

Input File

```
AFITDB
1
MCRS
M
MCRSCTRL
2
MCRSCTRL
8
Y
MCRSTITL
50
Y
1
MCRSCTRL
=
L
EENG
XXX
```

Result File

```
EENG755      THEORY OF COMPUTATION
  WAIV

EENG646      COMPUTER DATA BASE SYS
  WAIV

EENG608      POWER ELECTRONICS
  WAIV

EENG673      APPLICATIONS OF COMM TECH
  WAIV

EENG629      ELECTRONIC WARFARE I
  WAIV

EENG672      OPTICAL COMM & SIGNAL PROC
C WAIV

EENG589M     OPERATING SYSTEMS

EENG545      SOFTWARE SYS ACQUISITION
  WAIV

EENG451      SMALL COMPUTER SYS
  WAIV

EENG669      TECH APPLICATION SEM
  WAIV

-
-
-

EENG600      SEMINAR IN LOW OBSERVABLES
S WAIV

(171 TUPLES IN ALL)
```

Test Query Number Five

Roth Query

```
SELECT ALL FROM Member-of-Section WHERE
     (Section Number = 'GCS-85D') GIVING Temp1
JOIN Temp1, Student WHERE (Temp1.SSAN = Student.SSAN)
     GIVING Temp2
PROJECT Temp2 OVER (Student.Name)
     GIVING Temp3
```

Input File	Result File
AFITDB	Cxxxxxxx, APARECIDO F.
3	
SECT	Mxxxxx, RICHARD A.
M	
SECTCTRL	Hxxxxx, JENNIFER J.
1	
SECTCTRL	MAHONEY, KEVIN H.
8	
N	Txxxxxxx, BRIN A.
1	
SECTCTRL	Fxxxx, RICHARD E.
=	
L	Mxxxxx, RICHARD G.
GCS-85D	
XXX	Bxxxx, ALAN J.
SECL	
V	Cxxxxx, JOHN
SECLSECT	
SECTLKSE	Fxxxxxxxx, DAVID W.
LKSE	
2	Wxxxxx, STEPHEN L.
SECLSECT	
8	Hxxxxxxxx, CHARLES W. JR.
N	
SECLSTDT	Fxxxxx, JANICE H.
9	
N	Mxxxxxxx, BRUCE R.
0	
STDT	Sxxxxx, JAMES E.
M	
STDTCTRL	Wxxxxxx, JAMES A.
2	
STCTCTRL	Mxxxxxx, STEVEN C.
9	
N	Fxxxxxxx, DANIEL J.

Test Query Number Five (continued)

Input File	Result File
STDTNAME	
28	Qxxxxxx, TANVEER S.
Y	
1	Gxxxxxx, DAVID A.
STDTCTRL	
=	Wxxxx, STEPHEN A.
F	
SECLSTDT	Wxxxx, GREGORY B.
XXX	
	Wxxx, DONALD R. JR.
	Bxxxxx, ROBERT B.
	Wxxx, DONALD J.
	Mxxxxxx, RONALD A.
	-
	-
	-
	Wxxxxxx, ALEXANDER B.
	(40 TUPLES IN ALL)

(Last names were deleted for privacy purposes)

Test Query Number Six

Roth Query

```
SELECT ALL FROM Enrolled-In WHERE (Course Number = 'MATH555')
     AND (Quarter-Year = 'FA85') GIVING Temp1
JOIN Temp1, Student WHERE (Student.SSAN = Temp1.SSAN)
     GIVING Temp2
SELECT ALL FROM Member-of-Section WHERE (Number = 'GCS-85D)
     GIVING Temp3
JOIN Temp2, Temp3 WHERE (Student.SSAN = Member-of-
          Section.SSAN) GIVING Temp4
PROJECT Temp4 OVER (Student.Name, Enrolled-In Quarter-Year)
     GIVING Temp5
```

Input File	Result File
AFITDB	Hxxxxx, JENNIFER J.
4	FA85
SECT	
M	Mxxxxxx, KEVIN H.
SECTCTRL	FA85
8	
N	Txxxxxxx, BRIN A.
1	FA85
SECTCTRL	
=	Fxxxx, RICHARD E.
L	FA85
GCS-85D	
XXX	Mxxxxx, RICHARD G.
SECL	FA85
V	
SECLSECT	Cxxxxx, JOHN
SECTLKSE	FA85
LKSE	
2	Mxxxxxxx, BRUCE R.
SECLSECT	FA85
8	
N	Sxxxxx, JAMES E.
SECLSTDT	FA85
9	
N	Wxxxxxx, JAMES A.
0	FA85
STDT	
M	Fxxxxxxx, DANIEL J.
STDTCTRL	FA85
2	
STDTCTRL	Wxxxx, GREGORY B.

Test Query Number Six (continued)

Input File	Result File
2	FA85
STDTCTRL	
9	Wxxx, DONALD J.
N	FA85
STDTNAME	
28	Mxxxxxx, RONALD A.
Y	FA85
1	
STDTCTRL	Sxxxx, RONALD L.
=	FA85
F	
SECLSTDT	Bxxxx, MORGAN
XXX	FA85
VCQR	
V	Hxxxxxxxx, STEVEN A.
VCQRCODE	FA85
STDTLKCQ	
LKCQ	Mxxxxxx, THOMAS C.
3	FA85
VCQRCODE	
2	Dxxxxxx, FRANK W.
N	FA85
VCQRNMBR	
8	(18 TUPLES IN ALL)
N	
VCQRIDEN	(Last names were deleted for privacy purposes)
4	
Y	
2	
VCQRNMBR	
=	
L	
MATH555	
AND	
VCQRIDEN	
=	
L	
FA85	
XXX	

Test Query Number Seven

Roth Query

```
SELECT ALL FROM Enrolled-In WHERE (Course Number = 'MATH') AND
     (Quarter-Year = 'FA85') GIVING Temp1
JOIN Temp1, Student WHERE (Student.SSAN = Temp1.SSAN) AND
     (Student.Name < 'L') GIVING Temp2
SELECT ALL FROM Member-of-Section WHERE (Number = 'GCS-85D)
     GIVING Temp3
JOIN Temp2, Temp3 WHERE (Student.SSAN = Member-of-
          Section.SSAN) GIVING Temp4
JOIN Temp4, Course WHERE (Temp4.Course Number = Course.Number)
     GIVING Temp5
PROJECT Temp5 OVER (Student.Name, Enrolled-In Quarter-Year,
     Course.Title) GIVING Temp6
```

Input File	Result File
AFITDB	Oxxxxxxx, APARECIDO F.
5	FA85
SECT	MATH METHODS OF COMPUTER S
M	S WAIV
SECTCTRL	
8	Hxxxxx, JENNIFER J.
N	FA85
1	INTRO TO ADA
SECTCTRL	WAIV
=	
L	MAHONEY, KEVIN H.
GCS-85D	FA85
XXX	INTO TO ADA
SECL	WAIV
V	
SECLSECT	Txxxxxxx, BRIN A.
SECTLKSE	FA85
LKSE	INTERACOMMIVE COMPUTER GRA
2	A WAIV
SECLSECT	
8	Fxxxx, RICHARD E.
N	FA85
SECLSTDT	INTRO TO ADA
9	WAIV
N	
0	Mxxxxx, RICHARD G.
STDT	FA85
M	INTO TO ADA
STDTCTRL	WAIV

Test Query Number Seven (continued)

Input File	Result File
2	
STDTCTRL	Cxxxxx, JOHN
2	FA85
STDTCTRL	INTO TO ADA
9	WAIV
N	
STDTNAME	Fxxxxxxxx, DAVID W.
28	FA85
Y	APPLIED LINEAR ALGEBRA
2	WAIV
STDTCTRL	
=	Fxxxxx, JANICE H.
F	FA85
SECLSTDT	MATH METHODS OF COMPUTER S
AND	S WAIV
STDTNAME	
<	Fxxxxx, JANICE H.
L	FA85
N	INTERACOMMIVE COMPUTER GRA
XXX	A WAIV
VCQR	
V	Mxxxxxxx, BRUCE R.
VCQRCODE	FA85
STDTLKCQ	INTRO TO ADA
LKCQ	WAIV
3	
VCQRCODE	Sxxxxx, JAMES E.
2	FA85
N	MATH METHODS OF COMPUTER S
VCQRNMBR	S WAIV
8	
N	Sxxxxx, JAMES E.
VCQRIDEN	FA85
4	INTRO TO ADA
Y	WAIV
2	
VCQRNMBR	Wxxxxxx, JAMES A.
=	FA85
L	INTRO TO ADA
MATH	WAIV
AND	
VCQRIDEN	Wxxxxxx, JAMES A.
=	FA85
L	MATH METHODS OF COMPUTER S
FA85	S WAIV
XXX	

Test Query Number Seven (continued)

Input File

```
MCRS
M
MCRSCTRL
2
MCRSCTRL
8
N
MCRSTITL
50
Y
1
MCRSCTRL
=
F
VCQRNMBR
XXX
```

Result File

```
Fxxxxxxx, DANIEL J.
FA85
  MATH METHODS OF COMPUTER S
S WAIV

Fxxxxxxx, DANIEL J.
FA85
  INTRO TO ADA
  WAIV

Wxxxx, GREGORY B.
FA85
  INTRO TO ADA
  WAIV

Bxxxxx, ROBERT B.
FA85
  MATH METHODS OF COMPUTER S
S WAIV

Wxxx, DONALD J.
FA85
  INTRO TO ADA
  WAIV

Mxxxxxx, RONALD A.
FA85
  INTRO TO ADA
  WAIV

-
-
-

Lxx, JOHN M.
FA85
  MATH METHODS OF COMPUTER S
S WAIV

(31 TUPLES IN ALL)
```

(Last names were deleted for privacy purposes)

Appendix I: Summary Paper for

The Design and Implementation of a Relational

to Network Query Translator for a

Distributed Database Manangement System

ABSTRACT: The problem of translating global relational queries in a heterogeneous distributed database management system (DDBMS) to the Data Manipulation Language (DML) of the component database systems is examined. A specific approach toward a global query manager and the design of the global schema is proposed, and a query translation algorithm for the conversion of relational queries into the TOTAL network DBMS DML is presented. A set of sample queries translated by the program is presented and evaluated.

Introduction

Nearly all of the database management systems (DBMSs) in use today are one of three general models -- network, hierarchical, and relational. The advent of computer networks and the need for common information distributed across several locations has led to the development and use of the distributed data base management system (DDBMS). Most DDBMSs have been tailor-made for the user. However, several large companies and the government already have major investments in DBMSs that are often based upon the three different models. A way to tie these different DBMSs together under a common schema is needed. This problem was originally identified by Adiba (1) as the "communication and cooperation of heterogeneous databases".

The current situation at the Air Force Institute of Technology (AFIT) is a good example of this "heterogeneous distributed data base problem". AFIT has several DBMSs that it would like to tie together

using a local network. These DBMSs include TOTAL, a network DBMS, and several relational DBMSs, including INGRES, ORACLE, and dBase II.

Overview

The heterogeneous DDBMS area that was addressed by this effort was the development of translators for a "global" query language that could be used to retrieve information from any of the several different DBMSs. A single query language makes it much easier for a DDBMS user to retrieve information without having to know which DBMS holds the information and what the query language is for that particular DBMS. This particular system need be read-only, since the requirement for the majority of AFIT users is to gain access to the information, not to update it. The relational data model and relational algebra query language were used as the global data model and language.

This effort was divided into three stages: background research, the proposal of generic query translator algorithms, and the implementation of relational to TOTAL DBMS query translation software. After the system requirements were defined, the issues of data and query partitioning and the mapping between the various query languages were addressed. The final objective was to partially implement query translation software that takes the information provided by the relational query, and generates a program encompassing the equivalent TOTAL DML statements.

Previous DDBMS Development at AFIT

Previous AFIT thesis efforts by Imker (1982) and Boeckman (1984) involved the initial design and implementation of a DDBMS for a AFIT. Another effort by Jones (1984) specified that the relational model would be the global data model for the DDBMS. The main features of the resulting system are:

1. The DDBMS is a reconfigurable system. Nodes can be added to or deleted from the system and the central site can be relocated from one site to another. One node functions as the central site, which maintains the Central Network Data Dictionary (CNDD). The CNDD maintains information on all data stored throughout the DDBMS.

2. Each site maintains a Local Network Data Dictionary (LNDD) that maintains information on data stored at that site, and an Extended Network Data Dictionary (ECNDD) that maintains information on some of the data stored at other sites in the system. The ECNDD is permitted to grow only to a given size, usually smaller than the size of the CNDD. Each site is capable of handling queries and updates (i.e., translation software is at each node).

3. In the global schema, only First Normal Form will be required, although Third Normal Form will be used as often as possible. Duplicate keys (IMS and CODASYL) will not be allowed in underlying schema, and the partial replication of data must be supported, but full replication or no replication are the preferred choices.

Selection of DDBMS Approach

There were three major criteria for the heterogeneous DDBMS proposed for AFIT. The system should be: (1) able to access data that spans local databases without the user having to know the actual location, (2) able to deal with overlapping local schemas, and (3) reconfigurable to a point that eliminates critical nodes in the DDBMS network.

These requirements were matched against the six different approaches towards implementing a heterogeneous DDBMS that were defined by Katz in 1981 (10). Of the six approaches, the one that most closely matched the AFIT requirements was the Overlapping Database Prism. In this approach, the local database schemas are integrated into a global schema organized under a single global model. The user issues the query against this global schema in the DML of the global model, and the system then maps the query into a set of subqueries against the local databases. The actual location of the underlying data is transparent to the user.

Data Partitioning

One major problem is the range of data partitioning and redundancy that the DDBMS should be capable of handling. The importance of such a capability becomes apparent when the DDBMS must be able to decompose a global query into the appropriate (and most efficient) local DBMS queries.

In data partitioning, the global relation that the DDBMS user sees can actually be made up of several local relations, also known as fragments. According to Ullman (18:411), relations can be partitioned (fragmented) in two ways, vertically and horizontally. In both cases, the partitions of the database cannot be assumed to be disjoint. That is, data in the local relations could possibly overlap each other.

The other problem is redundant data. When integrating several existing DBMSs into a global DDBMS, there may be a large amount of redundant information present in the global system. For example,

information on a person could be in a personnel database file, a separate file in a student database, and another file in a payroll database. Most of the information in each database (name, address, age, etc.) is redundant. This falls into three categories: no redundancy, full redundancy, and partial redundancy.

Partitioning/Redundancy Classes

Ten different classes of partitioning and redundancy classes were defined. The classes that the system should handle are:

1. No Partitioning and No Redundancy. The given relation is unique. The data is stored as one relation at one site.

2. No Partitioning and Complete Redundancy. The global relation is composed of one local relation that is stored entirely at a single site as a complete relation. However, there is at least one complete duplicate copy of the information in the DDBMS.

3. Vertical Partitioning (No Redundancy). The data is not duplicated anywhere, but neither does the entire relation exist fully in one location. Each partition has all of the relation's tuples, but none contains all of the relation's attributes.

4. Vertical Partitioning (Partial Redundancy). Same as above, but the attributes of the global relation overlap each other within the local relations. Knowledge of which attributes exist in each local relation could prove useful in a projection (possibly eliminating the need for a join).

5. Horizontal Partitioning (No Redundancy). Each of the local relations possesses all of the necessary attributes of the global relation, but no single one of them contains all of the tuples of the global relation.

6. Horizontal Partitioning (Partial Redundancy). Each of the local relations possesses all of the necessary attributes, and some of the tuples exist in more than one local relation.

7. Vertical/Horizontal Partitioning (No Redundancy). The global relation is composed of two or more vertical partitions, one or more of which are further divided into horizontal

partitions. For proper recomposition of the relation, the union of the horizontal partitions must be accomplished before the join of the vertical partitions.

8. Vertical/Horizontal Partition (Partial Redundancy). Same as above, except that the horizontal partitions are partially redundant, and the vertical partitions could also be partially redundant.

9. Horizontal/Vertical Partitioning (No Redundancy). The global relation is composed of two or more horizontal partitions, one or more of which are further divided into vertical partitions. For proper recomposition of the relation, the join of the vertical partitions must be accomplished before the union of the horizontal partitions.

10. Horizontal/Vertical Partition (Partial Redundancy). Same as above, except that the horizontal partitions are partially redundant, and the vertical partitions could also be partially redundant.

Global Query Manager Functions

There are five main functions (7:34) of the DDBMS global query manager: (1) global data model analysis, (2) query decomposition, (3) execution plan generation, (4) query translation, and (5) results integration. These five functions are common to all DDBMSs, both heterogeneous and homogeneous, but the query translation and results integration pose special problems in the heterogeneous case.

Global Data Model. The global data model for the DDBMS was defined as the relational data model by Jones (9). The requirements of the global schema are further outlined in the next section.

Query Decomposition. Since the global model of the AFIT DDBMS is relational, the decomposition of the queries also follows the relational format. The global relational query will be decomposed

into a set of relational subqueries against the local databases, which are viewed logically by the system as relational schemas.

Query Translation. The approach taken by this thesis is that translation from the global relational query language to the local DBMS language will be done using the mapping approach, generating a procedural query that will produce the same result as the relational query. Since the user's view of the global schema is a relational one, the commands that they use should also be purely relational.

Results Integration. The recommendation and approach taken is to require the query translators to produce results in a canonical format (7:42). Each separate translator will return the results to the requesting node in the form of a relation. The advantage of this is that for "n" local DBMSs, only 2n translators are needed, and the addition of more systems to the DDBMS would not affect the current local DBMSs.

The Global Relational Schema

As previously noted, the relational model is used as the global relational model. The global model must be both schema and operation equivalent to the underlying models for proper translation of data and queries. A database is **schema-equivalent** to another database if there exists a mapping that maps the schema S_2 of the second database to the schema S_1 of the first database such that all constraints in S_2, that are essential in the context of the first database, can be preserved in S_1 (19:89). The resulting schema correspondences are depicted in

Figure I-1. If databases are schema-equivalent, and each operation on the first database can be mapped into a set of operations on the second database without loss of consistency, then the databases can be said to be **operation-equivalent.**

Elements of Global Relational Model	Corresponding CODASYL Elements	Corresponding IMS Elements
Domain	Occurrence of	Occurrence of
Attribute	Item Name	Field Name
Relation	Record-Type	Segment-Type
Foreign Key	Set-Type Link-Record-Type	Hierarchical Path

Figure I-1. Data Model Correspondence

Local Schema Constraints

The the underlying system schemas must conform to certain constraints in order that the integrity of the global schema can be maintained. In network databases, the schema must conform to the foreign-key constraint defined by Zaniolo (21:186). In hierarchical databases, the restrictions are: (1) all fields in the schema must be named uniquely, (2) each segment type must contain a hierarchical key, and (3) the overlying local and global relations generated from a hierarchical segment must contain the hierarchical keys of all the ancestor segments for that segment. These hierarchical keys propagated into the relation can be thought of as foreign keys of the relation. These restrictions were proposed by Vassilou and Lochovsky (19:90).

The Global Relational Query Language

Jones specified that the global language would be a relational query language, but would not necessarily be any presently designed language. As such, no specific query language is required by the query translation algorithms presented in the remainder of this thesis. By basing the query translator on the generic operations project, select, and join, it allows the potential use of any relational algebra or calculus based query language.

Query Translation to Hierarchical DML

The algorithms used were those proposed by Vassiliou and Lochovsky (19). The target system language is based on IMS, with GET-NEXT and GET-NEXT-WITHIN-PARENT as the basic commands, with recursive ability assumed for the programming language and system. These algorithms are neither complete nor optimal, but give a good start to dealing with the hierarchical translation problem.

Query Translation to CODASYL (Network) DML

The algorithms proposed are ones derived from the separate works of Katz (11) and Kuck (12). The processing selection algorithm produces an ordered list of records to be accessed, with the corresponding attributes and access characteristics. This is similar to the Iterative Query Language (IQL) proposed by Katz.

Once the most efficient network access sequence has been determined, the DML code for that particular access sequence must be generated. Generation of code was chosen over the use of set routines because of the flexibility that it offers in handling different

combinations of selects, projects, and joins. The DML generation algorithm used is based on the one proposed by Katz.

Query Translation to TOTAL (Network) DML

The main thrust of the thesis effort was toward developing a program to translate relational queries into the DML of TOTAL (a DBMS marketed by Cincom Systems, Inc.). TOTAL is a network database management system, but differs from the CODASYL proposal in several ways. Cardenas (4:218) describes the design concept of TOTAL as being 60 to 80 percent like CODASYL, but with a DML syntax similar to IMS.

TOTAL Data Management Language (DML)

The TOTAL DML is an extension to existing programming languages, consisting of a series of CALL statements to a TOTAL interface program known as DATBAS. There are three different types of DATBAS calls, each with its own parameter list. The first, using four parameters, is for signing onto TOTAL and for opening and closing the database schema. The second, using seven parameters, is for accessing Master datasets, and the third, using nine parameters, is for accessing Variable-Entry datasets.

TOTAL DML Generation Process

The translation process breaks down into three steps: creation of the structures used by the DML generator, ordering of the structures into the optimal processing sequence, and the generation of the source code with embedded TOTAL DML.

Dataset Structure Creation. The DML generation algorithms use as input a list of data structures similar to Katz's IQL. These structures contain the information necessary for the generation of proper DML code. This query information, obtained from the local data directory in the DDBMS, is as follows:

Query operation
Name of database--to get list of all datasets required
Name and type (master or variable) of datasets
Dataset Key
All required field names for each dataset
Size of all fields
Linkpath and reference names from the master dataset
    to variable dataset
Qualifier operators and operands (literal or fieldname)

TOTAL Access Ordering. Master data sets are accessed in only two ways; directly (for an equality comparison on a dataset key) and sequentially (all other cases). Variable data sets must be accessed sequentially through the chain beginning with the first or last data set in the chain. As such, the first dataset retrieved in query must be a Master dataset. Past this requirement, the problem of access ordering was not addressed in this effort. The most efficient access sequence is assumed to be present in the input.

Code Generation Algorithms

After the order selection is complete, the structures are input to the DML code generating routine. The main code-generating procedure, shown in Figure I-2, is derived from the CODASYL algorithm proposed by Katz (11).

```
procedure TOTGEN(i)
  IF (i > N) THEN DO
       Output all structure contents to output file
       return {to previous level}
  endif

  IF (Master Dataset) THEN
     IF (one-variable equality clause) AND
        (Clause is a Dataset Control Key) THEN

           create DML string
             "READM_i(<SearchKey_i>,<UserArea_i>)

               IF NotFound THEN GO TO LABEL(3*(i-1))
               IF (STATUS <> '****') THEN
                   GO TO ERRLABEL
               <Structure_i> := <UserArea_i>"

         IF (Structure_i.value(j) == qualifier)
                THEN (successful)
         IF (not successful) THEN GO TO LABEL(3*(i-1))"

        TOTGEN(i+1)

           create DML string
             "LABEL(3*(i-1)): "

     ELSE {sequential search - not dataset key}

           create DML string
              "Qualifier_i := 'BEGN'
               RDNXT_i(<Qualifier_i>,<UserArea_i>)
LABEL(3*(i-1)+2): IF QUALIFIER == 'end of chain' THEN
                   GO TO LABEL(3*(i-1))
               IF (STATUS <> '****') THEN
                   GO TO ERRLABEL
               <Structure_i> := <UserArea_i>"

         IF (Structure_i.value(j) == qualifier)
                THEN (successful)
         IF (not successful) THEN GO TO LABEL(3*(i-1)+1)"

           TOTGEN(i+1)

           create DML string
"LABEL(3*(i-1)+1): RDNXT_i(<Qualifier_i>,<UserArea_i>)
```

Figure I-2. TOTAL DML Generation Algorithm

```
                GO TO LABEL(3*(i-1)+2)
LABEL(3*(i-1)): "

   ELSE {Variable datasets - read chain in sequence}

            create DML string
               "<SearchKey_i> := <Dataset(i-1).KeyValue>
                READV_i(Reference_i,<SearchKey_i>,
                        <UserArea_i>)

LABEL(3*(i-1)+2): IF Reference_i == 'END.' THEN
                     GO TO LABEL(3*(i-1))
                  IF (STATUS <> '****') THEN
                     GO TO ERRLABEL
                  <Structure_i> := <UserArea_i>"

          IF (Structure_i.value(j) == qualifier)
                THEN (successful)
          IF (not successful) THEN GO TO LABEL(3*(i-1)+1)"

            TOTGEN(i+1)

            create DML string
"LABEL(3*(i-1)+1): READV_i(Reference_i,
                           <SearchKey_i>, <UserArea_i>)

                GO TO LABEL(3*(i-1)+2)
LABEL(3*(i-1)): "

   end procedure TOTGEN
```

Figure I-2 Continued. TOTAL DML Generation Algorithm

Subprocedures for DATBAS Calls. One of the more restrictive aspects of TOTAL (from a relational viewpoint) is that the DATBAS parameters are fixed. This means that for each dataset that is to be accessed by a translated query, there must be a corresponding declared DATBAS call with the unique parameter declarations. The only way to implement this and still allow any flexibility in the queries is to declare each DATBAS call within its own subprocedure, thus "hiding" that particular DATBAS call from all of the others. This means that a

separate subprocedure would need to be declared for each call to TOTAL.

TOTGEN DML Generation Algorithm. Once the variable and subprocedure declarations have been generated, the main body of the translated query program must be generated. This is handled by the TOTGEN procedure, shown in Figure I-2, which calls itself recursively in order to build the proper processing order into the program. In the figure, the subscript "i" signifies the declaration of variables for the i^{th} level of recursion, being the same as the number of the data set that is being retrieved. The angled brackets (<>) signify values (variables) that are generated for that particular call of TOTGEN.

Translation Software Host Machine and Language

Implementation of the translation software was done on the VAX-11 that also hosted the AFIT TOTAL DBMS. The translator algorithms were implemented using the C programming language. The DML generator portion of the program generates a Pascal language program with embedded TOTAL DML statements which, after being compiled and linked to TOTAL, actually execute the query.

Translator Limitations and Assumptions

The translation software was implemented using several limiting design decisions and assumptions.

1. The input is assumed to be the information passed to the local DBMS by the local data directory at the host node in the network.

2. The query translator program is capable of handling multiple databases resident on the TOTAL DBMS.

3. Queries are assumed to be input in the most efficient processing sequence. The immediate concern was to implement and test software that actually could translate queries.

4. The number of boolean qualifiers on each particular dataset was limited to two, with only the AND or OR logical connectives.

Translator Input and Output

There are two input files to the translation program, the query file and the database schema file. There are three output files, the generated source code, the object code (compiled and linked versions), and the query result file created by running the translated program. The following sections examine the two input files and the output result file.

Database Schema File. The information in the file is: (1) the size of the Schema Declaration (in characters), (2) the number of lines in the Schema Declaration, and (3) the schema information. The first two items of information are used to ease code generation, the schema size defining the buffer size for calls to TOTAL, and the line count allowing the proper formatting of the schema declaration. The actual schema information consists of the TOTAL options used and the dataset names of the database schema. The file that contains the schema information for the AFIT database is AFITDBSC.DAT (the database name concatenated with SC.DAT). As more TOTAL databases are tested, additional schema files will have to be created.

Query File. This file contains the query information that is returned by the LNDD in the DDBMS. The format of QUERY.DAT is:

Database Name (e.g. "AFITDB")
Number of Datasets (N) in Query
1st Dataset Information
-
-
Nth Dataset Information

The information for each dataset is essentially the same, containing the dataset name, field names and sizes, key field name and boolean qualifier information. There are differences in the formats between Master and Variable datasets, however. Variable datasets require additional linkpath and reference information that provides the specific association between the Variable dataset and its Master datasets.

Query Result File. The query result file is created when the query (generated program) is executed. The results are written in a simple format that separates each "tuple". The requested output from each of the datasets involved in the query is placed on a separate line, with another line separating each distinct data aggregate (tuple) in the result. For example, if a query requests a student name from one dataset, a course title from another, and the grade from a third, a resulting tuple might be:

Smith, John A.
Advanced Database Systems
A-

This query result file would then be transmitted back to the requesting DDBMS node for further processing, possibly to be joined or unioned with the results from a partitioned query to another database.

Processing Sequence

The processing sequence of the translator program, TRANS.C, basically consists of a series of passes down the array of dataset structures. First, the input from the query file is read in, building the array of datset structures for use by the remainder of the program. Each dataset is examined, and an access type classification of 1, 2 (both READM access), 3 (RDNXT), or 4 (READV) is assigned to each.

The code generation process now begins. After opening statements have been generated, the second pass down the dataset array checks the size of each dataset field and computes literal sizes in order to create the list of sizes for buffer-type declarations in the Pascal program. Another pass down the array creates record-types for query output. A fourth pass through the datasets and all fields generates the variable declarations for the program. The fifth pass down the dataset array generates one subprocedure (for the unique DATBAS call) for each dataset in the query. It is at this point that the database schema file is read. The sixth pass is made by the recursive module that generates the body of the Pascal program, with other, partial, searches of the array occurring as needed when computing the fields required in comparison qualifiers. The seventh complete pass is made when the recursion stops and output statements are generated.

At this point, the final code statements are generated and the generated source code file is closed. What remains now is to compile and link the program to TOTAL, execute the object code, and return the result file to the user.

Test Query Translations

Once the translation software was implemented on the VAX, a set of sample queries was run to test the operation and efficiency of the both the C translator/generation program and the generated Pascal program. The TOTAL database chosen for program testing was a subset of the AFIT Data Base (AFITDB), which is designed to handle the scheduling of classes, maintain student, faculty, and thesis information, and to track order information on textbooks. The portion of the AFITDB that was used consisted of four Master datasets and two Variable datasets, for a total of six datasets in the subschema. The datasets used were:

STDT - Student Master Dataset
SECT - Student Section Master Dataset
MCRS - Courses Master Dataset
MQTR - Quarters Master Dataset
VCQR - Variable Course-Quarter Dataset
SECL - Section Leader Variable Dataset

The specific test queries were chosen to represent a wide, but normal, range of queries that could be expected in the system. There were seven sample queries, involving from one to five of the six datasets in the schema. These seven queries were:

1. The courses that student "Mahoney" took in the Winter 1985 quarter, and is enrolled in for the Fall 1985 quarter.

2. The name and rank of all AFIT students with last names that begin with D, E, or F.

3. The name and rank for the student with student number "XXXXXX".

4. All course offerings of the Electrical Engineering Department.

5. The names of all the students in student section GCS-85D.

6. All the GCS-85D students that are enrolled in the Fall 1985 offering of course number MA555.

7. The names of all GCS-85D students with last names beginning with 'A' through 'J' that are taking a MATH deparment course in Fall 1985, with the associated course title.

Analysis of Query Translation and Execution

Stopwatch timing was done on all seven queries for each of the steps of query translation and execution: translation of the query file, compilation of the generated code, linking the code to the TOTAL DBMS, and executing the query. A graphical illustration of the query timing results is shown in Figure I-3.

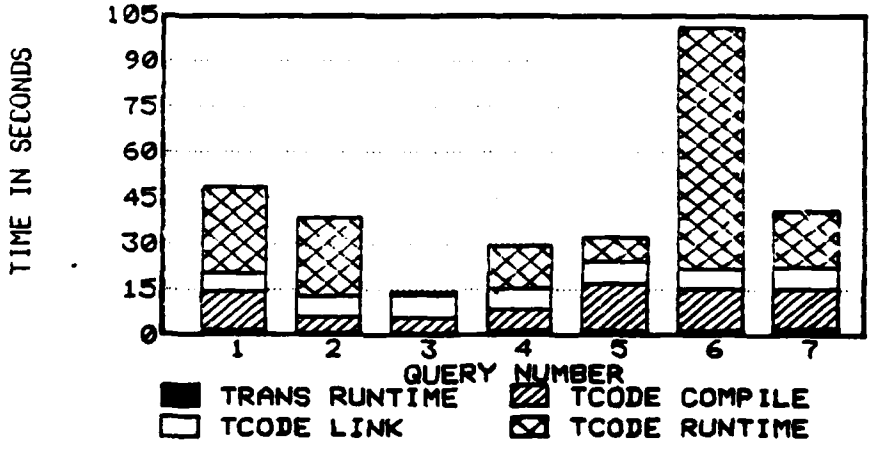


Figure I-3. Total Processing Time For Query Execution

Test Analysis

Most of the execution times did not reveal any surprises. Query 3, where a single record was retrieved by use of a key, was extremely fast, running in just over a second. The speed of this fastest query was expected. However, the slowest query, Number 6, was not anticipated. It ran for over 79 seconds, nearly three times as long as the next longest query. Complexity of the query might be the obvious answer, but this query did not involve the highest number of datasets (That was Query 7, which used five datasets, and which ran only 18.44 seconds). It appears that the difference lies in where the qualifications appear in the processing order of the query. In Query 6, the qualifications appeared in the last few datasets retrieved. .In Query 7, the qualification on the second dataset retrieved sharply reduced the amount of sequential searching required. The time it takes for the query translation process obviously differs by a wide margin from query to query. One cannot draw conclusions about the efficiency of such a query translator from a small sample, but some points can be drawn from these seven queries.

First of all, the length of the first three translation steps is fairly equal, even for the most involved queries. What varies widely is the time it takes to actually execute the query, but this is the case for all DBMS queries, not just translated ones.

It is apparent that the criteria for ordering the processing of datasets in a query (which was omitted from this partial implementation) should be expanded to include an analysis of where the qualifications lie in the processing order. Some knowledge about the

relative size of the datasets would also be useful. The combination of these two factors would help to reduce the execution time of the translated program.

Conclusions

Query translation in a heterogeneous distributed database system is a very real problem. The work of this thesis has indicated that this solution is not without its limitations. However, even if the translation of queries is not currently a particularly responsive solution, it is still the best approach short of converting the underlying local databases into a homogeneous system. It appears to be the only way that the "ad hoc" qualities of relational query languages can be preserved in a non-relational system. This thesis has shown that the translation of global queries into a different underlying DBMS query language is indeed possible. However, this effort has just scratched the surface. Future research into distributed databases, both heterogeneous and homogeneous, will hopefully continue to expand the body of knowledge concerning database systems.

Bibliography

1. Adiba, Michel and Portal, Dominique. "A Cooperation System for Heterogeneous Data Base Management Systems," Information Systems, 3 (3): 209-215 (1978).

2. Bernstein, Philip A. et al. "Query Processing in a System for Distributed Databases (SDD-1)," ACM Transactions on Database Systems, 6 (4): 602-625 (December 1981).

3. Boeckman, John G. Design and Implementation of the Digital Engineering Laboratory Distributed Database Management System, MS Thesis, GCS/ENG/84D-5. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1984.

4. Cardenas, Alfonso F. Data Base Management Systems. Boston: Allyn and Bacon, Inc., 1979.

5. Cardenas, Alfonso F. and Pirahesh, Mir H. "Data Base Communication in a Heterogeneous Data Base Management System Network," Information Systems, 5 (1): 55-79 (1980).

6. Date, C. J. An Introduction to Database Systems, (Third Edition). Reading, Mass.: Addison-Wesley Publishing Company, Inc., 1982.

7. Gligor, Virgil D. and Luckenbaugh, Gary L. "Interconnecting Heterogeneous Database Management Systems," IEEE Computer, 17 (1): 33-43 (January 1984).

8. Hevner, Alan R. and Yao, S. Bing. "Query Processing in Distributed Database Systems," IEEE Transactions on Software Engineering, 5 (3): 177-187 May 1979).

9. Jones, Anthony J. Analysis and Specification of a Universal Data Model for Distributed Data Base Systems, MS Thesis, GCS/ENG/84D-11. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1984.

10. Katz, Randy H. "Software Architectures for Heterogeneous Database Management," Proceedings IEEE COMPSAC 81. 33-42. IEEE Press, New York, 1981.

11. Katz, R. H. "Compilation of Relational Queries into CODASYL DML," Improving Database Usability and Responsiveness, edited by Peter Scheuermann, New York: Academic Press, 1982.

12. Kuck, Sharon M. A Design Methodology for a Universal Relation Scheme Implementation Via CODASYL, PhD Thesis. The Graduate College, University of Illinois at Urbana-Champaign, 1982.

13. Larson, James A. "Bridging the Gap Between Network and Relational Database Management Systems," IEEE Computer, 16 (9): 82-92 (September 1983).

14. P10-0002-01. TOTAL Users Manual for the VAX-11 Minicomputer, Release 2.0. Cincom Systems, Inc., Cincinnati OH, 1979.

15. Roth, Mark A. The Design and Implementation of a Pedagogical Relational Database System, MS Thesis, GCS/EE/79-14. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1979.

16. Takizawa, M. and Hamanaka, E. "Query Translation in Distributed Databases," Information Processing 80, edited by S. H. Lavington. Amsterdam: North-Holland Publishing Company, 1980.

17. Tsichritzis, D. C. and Lochovsky, F. H. Data Base Management Systems. New York: Academic Press, Inc., 1977.

18. Ullman, Jeffrey D. Principles of Database Systems, (Second Edition). Rockville, Md: Computer Science Press, 1982.

19. Vassiliou, Yannis and Lochovsky, F. H. "DBMS Transaction Translation," Proceedings IEEE COMPSAC 80. 89-96. IEEE Press, New York, 1980.

20. Wedertz, James A. The Design and Implementation of a Centralized Data Directory for a Distributed Database Management System, MS Thesis, GCS/ENG/85D-24. School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB, OH, December 1985.

21. Zaniolo, Carlo. "Design of Relational Views Over Network Schema," Proceedings A.C.M. SIGMOD Conf. 79. 179-190. (1979).

VITA

Captain Kevin H. Mahoney was born on 26 July 1955 in Stillwater, Oklahoma. He graduated from high school in Stillwater in 1973 and attended Oklahoma State University until May, 1977, at which time he enlisted in the USAF. Following Basic Military Training School at Lackland AFB, Texas, he attended the Computer Operations School at Sheppard AFB, Texas until September, 1977, when he was assigned to the Air Force Manpower and Personnel Center (AFMPC), Randolph AFB, Texas. While assigned to AFMPC, he was selected for the Airman Education and Commissioning Program (AECP), returning to Oklahoma State University in August, 1979. He received the degree of Bachelor of Science in Computing and Information Systems from Oklahoma State in May, 1981. In May, 1981 he entered Officer Training School (OTS) at the Lackland Training Annex, Texas, where he received his USAF commission in August, 1981. He was immediately assigned to Headquarters, Air Force Communications Command, Scott AFB, Illinois, where he remained until entering the School of Engineering, Air Force Institute of Technology, in May, 1984.

Permanent address : 624 Ute Avenue

Stillwater, Oklahoma 74075

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

AD-A164013

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; Distribution Unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) AFIT/GCS/ENG/85D-7	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION School of Engineering 6b. OFFICE SYMBOL (If applicable) AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State and ZIP Code) Air Force Institute of Technology Wright-Patterson AFB, Ohio 45433	7b. ADDRESS (City, State and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State and ZIP Code)	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT NO.
11. TITLE (Include Security Classification) See Box 19	

12. PERSONAL AUTHOR(S)
Kevin H. Mahoney, B.S., Capt., USAF

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Yr., Mo., Day)	15. PAGE COUNT
MS Thesis	FROM _____ TO _____	1985 December	251

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	Heterogeneous Distributed Database, Query Languages, Query Translation, Network Database Management System
09	02		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: THE DESIGN AND IMPLEMENTATION OF A RELATIONAL TO NETWORK QUERY TRANSLATOR FOR A DISTRIBUTED DATABASE MANAGEMENT SYSTEM

Thesis Chairman: Dr Thomas C. Hartrum

Approved for public release: IAW AFR 190-1.
16 JAN 86
LYNN E. WOLAVER
Dean for Research and Professional Development
Air Force Institute of Technology (AFIT)
Wright-Patterson AFB OH 45433

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☐ DTIC USERS ☐	UNCLASSIFIED

22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE NUMBER (Include Area Code)	22c. OFFICE SYMBOL
Dr. Thomas C. Hartrum	513-255-3576	AFIT/ENG

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

19.

A translation program was implemented for the translation of generic relational queries into the Data Management Language (DML) of the TOTAL data base mangement system. The objectives of this thesis were to propose and detail a method of supporting a global relational query language for a heterogeneous distributed database management system (DDBMS), design generic query translators for the translation of global relational queries into local hierarchical and network DML, and partially implement translation software for the conversion of relational queries into TOTAL DML.

The initial portion of the thesis presents an overall analysis of data partitioning, query decomposition and global query management in the DDBMS. Specific proposals are advocated concerning the specific approach to be taken toward a global query manager and the design of the global schema over current databases.

The second portion formalizes the assumptions and constraints present in the global relational model, and presents generic algorithms for the translation of relational queries into hierarchical and network DML.

The last portion details the implementation and testing of the relational to TOTAL translator. The approach used was to take information returned from the local data directory of the distributed database in response to a relational query, and "compile" that information into a generated Pascal program containing the TOTAL DML commands. Only the actual code generation portion of the software was implemented. Query parsing, query optimization, and results integration were not addressed. A set of sample queries translated by the program were presented and evaluated.

END

FILMED

4-86

DTIC